AD-A057 449 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2
HIGH-ORDER LANGUAGE TECHNOLOGY EVALUATION.(U)
OCT 76 T A DREISBACH, J L FELTY, I GREENBERG F19628-76-C-0225
UNCLASSIFIED IR-203-2 ESD-TR-77-125 NL

1 OF 3
AD
A057 449

LEVEL II

HIGH-ORDER LANGUAGE TECHNOLOGY EVALUATION

INTERMETRICS

This technical report has been reviewed and is approved for publication.

JOHN C. MOTT-SMITH
Project Officer
Techniques Engineering Division

JOHN T. HOLLAND, Lt Colonel, USAF
Chief, Techniques Engineering Division

FOR THE COMMANDER

STANLEY P. DERESKA, Colonel, USAF
Deputy Director, Computer
Systems Engineering
Deputy for Command & Management Systems

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-77-125	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) HIGH-ORDER LANGUAGE TECHNOLOGY EVALUATION		5. TYPE OF REPORT & PERIOD COVERED Final Report 6. PERFORMING ORG. REPORT NUMBER IR-203-2
7. AUTHOR(s) Timothy A. Dreisbach James L. Felty Ira Greenberg, et al		8. CONTRACT OR GRANT NUMBER(s) F19628-76-C-0225
9. PERFORMING ORGANIZATION NAME AND ADDRESS Intermetrics, Inc. 701 Concord Avenue Cambridge, MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division Hanscom AFB, MA 01731		12. REPORT DATE 15 October 1976 13. NUMBER OF PAGES 284
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

High-Order Language Technology
Semantic Specification Technology
PL/I
Computer Parallelism
Real-Time Systems
Operating Systems
Compiler Technology
Hardware Support of HOLs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is part of the Air Force Systems Command's high-order language (HOL) standardization program. The purpose of the program is to establish a coordinated position on standardization of high-order languages and support tools for use in the development of computer programs for Air Force systems. In order to produce viable, cost-effective, standardization recommendations, information on technologies related to HOL standardization is necessary.

→ cont on p 1473B

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

78 07 27 027

INTERMETRICS INCORPORATED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT Con't

~~The~~ This

Intermetrics' technology evaluation includes a technology forecast covering a number of areas of hardware and software with emphasis on their impact on high-order languages. In addition, the technology evaluation addresses the interface of high-order languages with operating systems and data base management systems, and also the impact of real-time technology on high-order languages with emphasis on communications applications.

A

1473B

(18) (19) ESD-TR-77-125

(6) HIGH-ORDER LANGUAGE TECHNOLOGY EVALUATION.

(9) Final Report.

(14) IR-203-2

(11) 15 October 1976

(12) 269p

(10) Timothy A. Dreisbach,
James L. Felty,
Ira Greenberg,
Robert E. Hartman
Alan Kramer
[redacted], Roth
[redacted] Weissman

Intermetrics, Inc.
701 Concord Avenue
Cambridge, Massachusetts 02138

Prepared for:

(15) Contract F19628-76-C-0225
Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, Massachusetts 01731

The purpose of this report is the exchange of ideas. It does not represent an official position of ESD or of the Air Force.

388 863

This report has been prepared for the Electronics Systems Division, Air Force Systems Command as CDRL Item A004 under Contract Number F19628-76-C-0225. Mr. John Mott-Smith is the Contract Technical Monitor for the Air Force. Dr. Larry Weissman is the Program Manager for Intermetrics, Inc.

The authors wish to acknowledge the advice and technical guidance of Dr. James S. Miller and the editorial and technical help of Dr. Benjamin M. Brosgol during the preparation of this report. Special thanks are due Miss Judith Haigh and Mrs. Deborah Gleason for their typing efforts.

TABLE OF CONTENTS

0.0 INTRODUCTION

This report is part of the Air Force Systems Command's high-order language (HOL) standardization program. The purpose of the program is to establish a coordinated position on standardization of high-order languages and support tools for use in the development of computer programs for Air Force systems. In order to produce viable, cost-effective, standardization recommendations, information on technologies related to HOL standardization is necessary.

Intermetrics' technology evaluation includes a technology forecast covering a number of areas of hardware and software with emphasis on their impact on high-order languages. In addition, the technology evaluation addresses the interface of high-order languages with operating systems and data base management systems, and also the impact of real-time technology on high-order languages with emphasis on communications applications. Section 1 is an executive summary of the report.

One important area that can have an impact on language standardization is that of software technology in the fields of programming languages and compilers. Part I A deals with various aspects of this area. Section 2 discusses compiler technology and emphasizes compilation techniques, compiler construction, cross compilation, and reliability, especially as they affect language standardization. Section 3 covers semantic specification techniques. An important aspect of language standardization is the ability to specify through such techniques a definition of a standard language in a precise and unambiguous fashion. Section 4 deals specifically with the area of high-order language technology. Forecasts are made both with regard to research being done in programming languages and with regard to changes

to existing standard languages. In addition, trends in establishing languages as standards are discussed. Section 5 discusses the suitability of a large extremely general purpose language for Air Force applications. The particular language discussed is PL/I.

Another important area that can have an impact on language standardization is that of hardware technology. Part I B deals with various aspects of this area. Section 6 deals specifically with forecasted improvements in hardware technology that can be used to support features directly in high-order languages. Section 7 covers computer parallelism both on the architectural level and on the language interface level and discusses the relationships between parallelism as handled in the hardware, the operating system, and the high-order language. Section 8 deals with the question of input/output facilities and discusses the tradeoffs between various ways of handling them in programming languages.

A final area that can have an important impact on language standardization is that of software technology in application areas that use high-order languages. Part II deals with three application areas of particular relevance to the Air Force. Part II A deals with data base management systems. Section 9 forecasts improvements in data base management system technology and assesses the impact these will have on the interface with high-order languages. Section 10 discusses the impact of data base management systems on HOL portability.

Part II B deals with operating systems. Section 11 forecasts improvements in operating systems specifically as they relate to high-order languages and integrates the forecast by discussing the interface between operating systems and high-order languages, including both facilities needed in languages to implement operating systems and also facilities needed in high-order languages (and job control/command languages) for application programs to interface with the operating systems.

Part II C deals with real-time systems. Section 12 discusses requirements for real-time languages and evaluates ten languages against these

requirements. Section 13 discusses hardware support of real-time languages. Section 14 forecasts hardware and software improvements that will occur in the area of communications and evaluates the impact of communications hardware and protocol standards on software portability and language standardization.

1.0 EXECUTIVE SUMMARY

This section summarizes the important points made in the body of the report. The various points are focused to indicate the impact they will have on Air Force HOL Standardization. All the points are keyed to the sections of the report in which more detail and supportive material can be found. In addition, each section of the report concludes with a summary of the material contained in that section. The time-frame terms used are the following -- short-term future: less than 3 years; medium-term future: 3 through 7 years; long-term future: greater than 7 years.

The major results of this technology evaluation are as follows:

(1) In the short-term future, computer hardware costs will continue to decrease significantly and the performance will continue to improve significantly. Consequently, less emphasis will be needed on efficiency of the software and it will be possible to devote more attention to producing software that is well-written, readable, reliable and easily modifiable. This shift in emphasis has a direct bearing on what languages should be chosen as standards. The trend will be towards simple languages whose features encourage the production of software having the above-mentioned qualities (Section 6).

(2) Despite the firm entrenchment of languages like COBOL and FORTRAN, languages have been developed (e.g., PASCAL and SIMULA 67) that encourage the writing of clear readable programs and further research is proceeding to develop even better languages (e.g., CS-4, CLU, and ALPHARD). The major advances of such languages are in the areas of reliability (with such capabilities as strong typing, useful redundancy, and exception handling) and structuring (with such capabilities as structured control flow, functional abstractions, and data abstractions). The main empahsis of the research will be on data abstractions, a concept that originally appeared in SIMULA 67 and which has been further refined and developed in CS-4, CLU, and ALPHARD (Section 4).

(3) Although FORTRAN and COBOL have evolved throughout their lifetimes and will continue to do so, they have remained and will continue to

remain behind the state-of-the-art. They will never be able to evolve, however, to the point where even today's languages currently are while still maintaining upward-compatibility with their previous versions. Consequently, to the extent that technical considerations of the language prevail, FORTRAN and COBOL should not be used for new Air Force projects. Of the languages developed more recently and in current use (e.g., PASCAL, JOVIAL, PL/I, SIMULA 67), none stands out as being vastly superior to the others in all respects (Section 12). Consequently, current use of such languages can continue as-is for the short-term future although it is advisable to stop the proliferation of languages by choosing an interim standard on other than purely technical grounds. In the next five to ten years, however, research in such areas as data abstractions will have progressed to the point where either significantly better languages will emerge which should replace existing languages, or else the infeasibility of such concepts will have been demonstrated.

(4) In the long-term future, computer architectures will be developed (e.g., descriptor-based architectures) that will greatly facilitate the implementation of such languages as CS-4, CLU, and ALPHARD. Language designers will not (and should not) be constrained to design and implement languages for machine-oriented-language machines which manufacturers have heretofore provided. There is no longer justification for performing language design and hardware architecture design separately. In the future not only will language and hardware architecture design be done together, but it will be the language design that will strongly influence the hardware design rather than vice versa as is currently the case. Research has already been done in language-directed machine design, but the impact of such an approach is still a long way off (Section 6).

(5) In the short term, language implementation and hence standardization and control will be facilitated by advances that have been made in automatic compiler generation techniques. These advances have been primarily in the front-end of the compiler (scanner, parser, error-handler, machine-independent optimizer). Research is being done in code generation

and in intermediate language standardization, but significant results are not likely in the short term. Automated compiler construction aids (even when limited to the front-end of the compiler) will have a significant impact on language standardization because the task of generating the large number of compilers required will be simplified and kept under control by the ability to supply automatically-generated sections of the compiler. In addition, the reliability of such compilers will be increased. However, these methods will also make it easier for more HOLs to appear and hence complicate HOL standardization unless proper administrative controls are enforced (Section 2).

(6) Language standardization and control will also be facilitated in the long-term future through the use of formal semantic definitions. Such definitions will provide configuration control tools for languages. In addition, they will form an integral part of programming language design by aiding in the analysis of particular language constructs as well as the overall design. Before formal semantic definitions can have a major impact, however, more research is needed to determine which conceptual approaches and specific techniques are the best and also to make the techniques more understandable and usable to the general programming community. Similar research needs to be conducted before accurate forecasts can be made regarding the practicality of their use in automatic test case generation for the certification of standard compilers (Section 3).

(7) Federated computer systems and networks will continue to increase in prominence in the future. Along with this, research will proceed in formalizing the interactions of multiple processors in a network in order to utilize the capabilities provided to their fullest potential. In the long-term future, such research will impact programming language design, since HOL features will be needed to support these multiple processor interactions. However, it is too early now to tell what these features might be (Section 7).

(8) Along with the increased use of federated computer systems and networks will be the increased use of data base management systems utilizing both distributed processors and distributed data bases. Additionally, special back-end DBMS processors will be developed. Within six years, the CODASYL DBTG proposals for the HOL-DBMS interface will emerge as the industry standard. Consequently, any standard HOL used by the Air Force must include a DBMS interface consisting of a Data Manipulation Language and sub-schema Data Definition Language. The problems of transporting data bases themselves, through the use of automated data translation systems, will remain unsolved for at least eight to twelve years (Sections 9, 10).

(9) Operating system technology will have a two-fold impact on language standardization. The first area of impact involves providing features in a language for implementing operating systems. The second area involves the user interface with the operating system. Work is being done both in defining programming language-operating system interfaces and in defining job control/command languages. Within ten years standards for these are likely to emerge. Standardization of the job control/command language, however, will have no impact on HOL standardization (Section 11).

(10) Real-time systems and communication systems will not have a major impact on HOL standardization. Both software and hardware requirements for real-time systems differ more in degree than in nature from other systems. Hardware improvements, especially protocol processing by micro-computer and single-chip interfaces, will increase the desirability of using HOLs for communication systems in the short-term future (Sections 12, 13, 14).

PART I.

HOL-RELATED SOFTWARE AND HARDWARE TECHNOLOGY

A. Software

2.0 COMPILER TECHNOLOGY

A discussion of HOL standardization of necessity entails a discussion of compilers. It is through a compiler that the behavior of programs written in a language is actually seen. In fact, the compiler itself often becomes a de facto "standard" for a language, arbitrating subtle questions concerning the language by virtue of the code that the compiler happens to produce. A compiler is obviously not an ideal standard for defining a language, especially when one wishes to write a second compiler (for a different machine, say) that implements exactly the same language.

An Air Force HOL standardization program, involving many compilers for many different machines, will place some rather stringent requirements on compiler technology and compilers themselves. Some of these requirements are:

1) It must be possible to write a compiler that implements all the features of the language.

2) One should be able to demonstrate that a compiler exactly implements the language for which it was designed. The advantages of any standardization program are reduced if differences exist among a language's several compilers. Any deviations that a given compiler makes from the language standard will cause programs to have (perhaps subtle) unpredicted behavior and will inhibit transportability of programs from one compiler to another.

3) It must be possible to construct compilers so that the language may be used on many different computers. It will be necessary to implement many compilers for a variety of host and target machines. It would be desirable to decrease the effort

and cost involved in creating or transporting these compilers and thereby increase the availability of the language.

4) It must be possible to construct each compiler such that it meets constraints such as time for construction, size, and speed of compilation. It would be unfortunate to standardize on a language without having given proper thought to these constructability and efficiency issues. Also, any technological advances that improve compilers for a particular language would add to the desirability of that language.

The following subsections address four current areas of compiler technology as they affect these requirements that HOL standardization places on compilers. These areas are:

Compilation Techniques -- techniques which improve the performance of compilers by improving their internal design.

Compiler Construction Techniques -- techniques which facilitate the actual writing of a compiler.

Cross Compilation -- a technique which enables a language to be used for a computer which does not host a compiler for that language, and

Reliability -- the assurance that the actual behavior of the compiler in fact matches its intended performance.

For each area, the current state of the art is assessed, and the direction of future technology is predicted.

2.1 Compilation Techniques

Since compiler performance is a factor influencing the choice of a language for standardization, it is relevant to identify advances in compilation techniques and to attempt to predict future trends in this

area. Particular topics to be considered are parsing, error handling, and optimization.

2.1.1 Parsing

The parsing (or syntactic analysis) phase of compilation has been studied extensively for about fifteen years. The reasons for the attention it has received are 1) the subject is mathematically formalizable and quite tractable (especially when compared with the machine-dependent parts of a compiler); and 2) there is a need for an efficient parsing technique which is applicable to a general class of context free (i.e., BNF) grammars. Although reason 1) is still viable, the requirement mentioned in 2) has been fulfilled fairly successfully -- primarily with the development of LALR [LaLonde 1971, Horning 1974a] and LL [Rosenkrantz and Stearns 1969, and Griffiths 1974] grammars.

As a result, it is likely that the attention paid to parsing techniques will diminish in the future, concentrating primarily on "tuning up" existing approaches as opposed to developing new ones. A survey of papers at recent technical conferences seems to support this claim: e.g., at the 8^{th} Annual Symposium on Theory of Computing (May 1976), only one paper out of thirty [Graham et. al. 1976] dealt with parsing. Another reason for the likely decline in work on parsing is the emergence of new areas in theoretical computer science which are attracting researchers who previously have dealt with parsing. Two such areas currently receiving widespread attention are formal proofs of program correctness and the theory of computational complexity.

During the coming years, parsers based on LALR and LL grammars will likely become dominant. Some reasons for this are 1) both methods allow better error handling than other approaches, and 2) parsers for both types of grammars can be constructed automatically fairly efficiently. Two compiler generating systems that have incorporated LALR parsers

are XPL and AED [Anderson 1973].

As a historical note, it should be mentioned that the most influential single development in parsing was probably Knuth's work on LR(k) grammars [Knuth 1965]. This paper established a number of significant theoretical results (e.g., the relationship between LR(k) grammars and deterministic pushdown automata) and established the framework for later practical results in parsing techniques. This framework was fleshed out most successfully by DeRemer in his thesis on Simple LR(k) grammars [DeRemer 1969] and (building on DeRemer's work) by LaLonde in his thesis on LALR(k) grammars [LaLonde 1971]. The deterministic languages (the largest practically recognizable class of languages) is precisely the class defined by the LALR(1) grammars. Although the LL languages are a subset of the deterministic languages, in practice this is not necessarily a limitation "since programming languages do not seem to fall in the gap between LL(1) languages and deterministic languages" [Horning 1974a].

2.1.2 Error Handling

In recent years a large effort has been focused on error handling -- i.e., the ability of a compiler to detect, diagnose, and perhaps "recover from" errors it encounters when parsing a program [Leinius 1970, Levy 1971, Peterson 1972, Graham and Rhodes 1975]. Recent advances include 1) the use of context both before and after the error point to perform a more accurate diagnosis, 2) an improved ability to indicate that the error occurred in the text that was parsed before the problem was detected, 3) an improved ability to process input with closely spaced errors, and 4) systematic approaches to error recovery (as opposed to ad hoc methods) which allow automation to be used.

Error recovery is particularly important because it is used as a verification tool. In this respect, the ability to systematize and

automate error recovery is very desirable. The advances that have been made will begin being used in compilers in the near future. An example of a promising approach is that described in [Graham and Rhodes 1975].

2.1.3 Optimization

Since the advent of high order languages, the desire for HOL compilers to produce "good" code has been strong. Much of the work on the first Fortran I compiler was devoted to optimization of the object code. "...The group had one primary fear. After working long and hard to produce a good translator program, an important application...of the sort that FORTRAN was designed to handle...though well-programmed in FORTRAN...would run at half the speed of hand-coded version." [Backus and Heising, 1964]. It was correctly foreseen that the language's acceptance would rest on the compilers' ability to avoid such situations. The continued evolution of HOLs influenced by a growing emphasis on readability and reliability, has placed a greater distance between the source and object code. As this distance increased, so did the requirement for improved optimization techniques, since efficiency was still an important goal. In recent years, more powerful and efficient methods have begun to appear. Their inclusion in commercial compilers can be foreseen in the near future.

Generally, optimizations are seen as being applied on two distinct levels: local and global. For the purposes of this discussion we will introduce a third level which we call micro. In this category of micro optimizations we include all those whose purpose is the optimal use of a particular machine's facilities. This enables us to isolate what are usually considered machine-dependent local optimizations from those which are machine-independent.

Micro optimizations are those which are not expressible as source code transformations. Efficient register utilization and the use of special machine instructions both fall into this category. The first of these is an extremely well-explored topic and today is considered by many a requirement of any "good" compiler. The techniques to be applied for its realization can be found in [Nakata 1967] , [Radzeijowsky, 1969] and [Sethi and Ullman, 1970].

The optimal utilization of a particular machine's instruction set is a much more complex problem. While ad hoc solutions to specific cases have been used in almost every compiler, the search for a general theory of code generation is just beginning.

Several languages have been designed in such a manner as to partially circumvent this problem. Two approaches are: 1) to include special operators which simplify the identification of cases where particular machine instructions may be used, and 2) to enable the programmer to explicitly specify assembly language code within his HOL program. An example of the first is the plus-assignment operator "+:=" in Algol 68. This operator is directly translatable into an add-to-memory instruction on many machines. Although these two techniques will continue to be used in the future, they each have drawbacks. Extending the first approach to its limit vastly complicates the language and still does not allow for all possible future developments in machine design. The embedding of assembly language code is a powerful facility, but it detracts from the readability and machine-independence of the program.

The distinction between local and global optimizations rests in the scope of information considered in their application. Local optimizations are applied to regions of contiguous code sequences containing no conditional statements. Such regions are commonly called program blocks. Optimizations within program blocks include: 1)removal of useless assignments, 2) elimination of common sub-expressions, (3) reduction in strength, and 4) compile-time evaluation of constant expressions.

Reduction-in-strength optimizations are those which replace time-consuming operations by more efficient ones. An example of this is the replacement of an exponentiation usually to the second or third power, by multiplications.

Many optimizations on both the micro and local levels alter the order of evaluation for certain expressions. In some cases such alterations are accomplished through the application of the commutativity rules

of arithmetic. While it has been shown that computer arithmetic does not always adhere to these rules due to its finite precision [Knuth, 1969], many language standards still permit their use (e.g. FORTRAN USAS X3.9-1966). However, this discrepancy can significantly modify the result and only in languages that require the specification of needed precision for all fixed and floating-point variables can it be identified.

Although true optimization is possible, within program blocks, it is not possible over more global regions [Aho and Ullman, 1973] . This fact stems from results on the undecidability of program equivalence [Aho 1970]. To avoid problems of undecidability, global optimizations are accomplished by applying equivalence-preserving transformations to the program.

Global code optimization is at present a quickly evolving field. New results have recently been emerging which permit more efficient and general algorithms to be applied. Some of the most recent work in this field is contained in[Graham and Wegman, 1976, Taniguchi and Kasami, 1976, Morel and Renvoise, Allen and Cooke, 1976 and Kam and Ullman, 1976].

These papers present algorithms which for well-structured programs operate in almost linear time. What this means is that the overall complexity (in terms of time of execution) for these algorithms approaches that of deterministic parsing. These papers deal with the global elimination of redundant computation and useless assignments, and the movement of invariant computations out of loops.

While these new algorithms have yet to be incorporated in any compiler, their use in the near future is a certainty. The work of Morel and Renvoise is possibly an exception to the above statement, inasmuch as it has been done in conjunction with the LIS language [Ichbiah et. al., 1974], but a full report has yet to appear in the literature.

In summary, optimization has always been seen as critical to HOL acceptance. The most important recent developments in optimization have

been in the area of global code improvement. We can expect that those advances will impact on compiler development in the near future. Their greater capabilities will facilitate the acceptance of new, more readable and reliable HOLs.

2.2 Compiler Construction

One method of writing a compiler is to start from scratch and write it by hand. Another approach is to combine hand coding with the tools of a compiler writing system. A compiler writing system can contain three types of tools: 1) programs that are used to generate parts of the finished compiler from descriptions that are provided, 2) precoded routines that actually form part of the finished compiler, and 3) routines used to simplify programming but which are not part of the compiler.

The various methods of compiler construction are discussed in the paragraphs below. As these methods improve and make compiler construction easier, they will also make it easier for more HOLs to appear. They could therefore complicate HOL standardization unless proper administrative controls are enforced.

2.2.1 Programming by Hand

There has been a great deal of activity in the areas of programming and project management in the past five or six years. People realized that these areas were not only more difficult than previously thought, but also were becoming very costly. A wide variety of tools and techniques have been proposed to remedy this situation.

The theories that these ideas are based on include reduction of complexity and control of information through abstraction, modularity, hierarchy, structure, redundancy, and complete interface specification. Examples of actual methods and tools for design, programming, and project management will be discussed in Section 2.4, Reliability. In fact,

creating reliable and maintainable software was a primary goal of this research.

One approach that stands out from the others is modularity. Tools and methods that support modularity are very important because of its centrality in the compilation process. One important place to divide a compiler is between the language dependent and machine dependent parts. This kind of modularity enables a compiler to be more easily available for different languages and different machines. By changing the language dependent part, a new language can be compiled, and by changing the machine dependent part, code can be generated for a new machine. Also, by splitting the compiler into several passes, implementors can make the compiler satisfy constraints on storage.

From their use on actual projects, these methods have been shown to be successful in improving both software manageability and cost. The use of these or similar approaches will increase and become widely used during the coming years.

Another programming technique that enhances availability is parameterization. Parameterization can be used to simplify the task of: 1) adjusting machine oriented aspects of the compiler, 2) adjusting compiler limits such as table sizes, and 3) selecting various compiler options such as the inclusion of an optimization phase.

Finally, effective programming of a compiler by hand requires a wise choice of programming language. Horning [1974b] states the point well:

> The choice of an appropriate language can reduce the cost of compiler by an order of magnitude. This is due to two effects: in the right language, the source text of the compiler will be much shorter, and a suitable language will reduce the opportunities for error and assist in the detection of errors that do occur.
>
> We may identify a number of requirements of a language suitable for writing compilers:
> - it must be easily readable and understandable
> - it must have appropriate data objects (e.g., Booleans, integers, characters) and operations on them

- it must have simple yet powerful control and data structures (e.g., iteration, vectors, selection)
- it must contain enough redundancy for substantial compile-time checking
- it must support modularisation (e.g., by means of macros, procedures, and data type definitions), preferably with secure separate compilation
- it must allow separate and checkable specification of module interfaces
- it must map efficiently into machine code. Assembly languages are unsuitable on almost all accounts, but many high-level languages are also unsatisfactory.

2.2.2 Compiler Writing Systems

A compiler writing system generates as output portions of a compiler. One advantage of compiler writing systems is that they aid in the creation of compilers that exactly implement a given language. If a system inserts a precoded routine into a compiler or uses a generating program to create part of a compiler, then that part would be identical in different compilers for a language. Since this part would have to be verified only once, it would simplify the verification and validation of the entire compiler. Also, an automatic generation program of proven reliability can decrease the time and cost to verify the part of the compiler written by that program.

Compiler writing systems also make it possible to construct compilers more quickly and at less cost.

One of the best compiler writing systems is that of [Lecarme and Bochman 1974]. The system contains a scanner generator, a parser generator, a simple error handling generator, and a test case generator. These test cases are useful for checking the grammar description and partially testing the semantic actions. The writing of semantic routines is aided by semantic attributes and a set of utility routines. The entire system is written in PASCAL, semantic routines are written by the user in PASCAL, and the compiler output by the system is written in PASCAL. Some other noteworthy systems are AED [AED 1969], JOCIT [Dunbar 1975], SPLIT [Anderson 1973], and XPL [McKeeman et. al. 1970].

Systems such as those mentioned above can be used at the present time with beneficial results. The use of compiler writing systems will increase in the future, especially for the construction of parsers and error handlers. This increase will take place slowly.

Work is currently being done on automating the code generation phase of the compiler, but "it will be some time before it is possible to simply enter a description of the target computer and produce a quality code generator" [Dunbar 1975]. The major problems are the interfaces to operating systems and differences among hardware architectures. This last point is especially true in view of 1) the recent explosion of micro and mini computers and 2) the wide variety of special purpose machines used by the Air Force.

One area where promising research is being performed is the automation of context sensitive parts of programming language semantics. One person doing such work is Gregor Bochman at the University of Montreal. If progress continues as expected, results will be available in a few years.

2.2.3 Intermediate Languages

Intermediate languages are used as interfaces between the phases of a compiler. One important interface is that between the language and machine-dependent parts of a compiler. There is question as to whether the intermediate language used for this interface should be standardized.

Standardization would provide several advantages. One is that availability would be improved. If the intermediate language were designed so as to be easy to implement, then compilers could be transferred to a new computer relatively quickly by writing a translator or interpreter for the intermediate language. One ongoing project along these lines is the Janus system [Poole 1974]. Also, it would be possible to connect various front-ends for different languages to back-ends for different machines in order to create a variety of compilers quickly. One effort

in this direction was the CWS project of McDonnell Douglas [Anderson 1973]. Unfortunately it is currently halted because of a lack of funds [Jelinski 1976]. These approaches would also be useful in reducing compiler construction costs.

The disadvantages of standardizing the intermediate language are related to efficiency. Will the speed and size of the compiler as well as the object code be good enough? It is difficult to find one intermediate language that will efficiently span a variety of language features as well as machine architectures. This consideration is especially important for the Air Force because it has a wide range of language requirements and uses many specialized machines.

2.3 Cross Compilation

Cross compilation refers to the process of compiling for a target machine that is different than the machine upon which the compiler is hosted. There are several reasons why one would want to do this.

A compiler that operates in this manner is one stage in a bootstrapping process. By recompiling this compiler for the target machine a new compiler would be produced that would be hosted on and produce code for the target machine. If the compiler were written in the language it was designed to compile, it could in fact be used to compile itself for the new machine.

Sometimes one machine holds some advantages over another machine. Even though it is necessary to produce code for the second machine, it may be desirable to have the compiler on a different machine. Possible advantages would be 1) greater speed, 2) extra secondary storage, 3) more available computer time, and 4) lower cost of operation. It may even be impossible for the target machine to host the compiler. For example, the target computer may have insufficient primary or secondary storage.

Another reason for using cross compilation is when the development and usage tasks are separated and different machines are involved. In

fact, it may be necessary for a program developed in one location to be used on many different machines. Also, it may not be desirable to have a compiler on the target machine because it would be too wasteful or there would not be a need for recompilation.

For the reasons stated above, cross compilation will continue to be an important technique.

One implicit assumption of cross compilation is that the design of the compiler is modular so that the machine-dependent portion can be replaced easily. This may be a good situation in which to take advantage of a standard intermediate language. One additional point to keep in mind is that all support software must also be transferred to the target machine as well as the program being compiled.

2.4 Reliability

The reliability of any compiler is of vital importance. Unless one is guaranteed a reliable compiler, all programs processed by it must be suspect. This concern is even stronger with compilers for a standard language because 1) of their widespread use, and 2) any error can cause the compiler to deviate from the standard.

A compiler's reliability, just as that of any large software system, can be enhanced through the use of proper project management and programming techniques. In addition, the use of both tools and techniques specifically geared to compiler development can increase the reliability of the finished product.

The following sections will first discuss techniques which are important to software reliability in general and then those which are unique to compilers. Each discussion will be further divided into techniques applicable during and after implementation.

2.4.1 Software Reliability

2.4.1.1 Attention to Reliability During Implementation

Building reliability into software during its creation has received a large amount of attention in recent years. The idea is that by spending effort on reliability as the software is built, the overall costs associated with the project will be reduced and that an inherently more reliable product will be produced. With respect to the overall quality of a program it is better to be concerned with reliability during construction than after construction.

A wide variety of tools and techniques have been studied in this area. The tools include formal specification languages and properly designed programming languages, and the techniques include program design, program structuring, and project management.

Formal specifications are important with respect to reliability because they provide complete information to guide software design. Also the construction of a formal specification often uncovers overly complex situations and areas that have not been completely thought out. Semantic specification languages are covered in depth in Section 3.

The choice of a programming language also affects the reliability of software, including compilers. In recent years many features have been proposed which will have a positive effect on reliability. Some of these features are based on the idea of controlling complexity, such as the flow of control and data structuring features in PASCAL [Jensen and Wirth 1975]. Other features try to reflect design methodologies such as

abstraction [Brosgol et. al. 1975, Liskov 1976, Wulf 1974a] and modularity [DeRemer and Kron 1975]. Another approach is to design features that are less error prone. Research has shown that some notations and constructs prove to be inherently more error prone than others [Gannon and Horning 1975]. The effect of programming languages on reliability is discussed further in Section 4.

The reliability of software may be enhanced by modern software techniques. Some of these are "top-down" program design [Wirth 1971], use of levels of abstraction in programs [Dijkstra 1972a] , and modular decomposition of programs [Parnas 1972] . The use of top-down design has also been advocated because of its usefulness in software verification [Smith 1975]. Project management includes project organization methods like chief programmer teams [Mills 1971, Barry and Naughton 1975] and egoless programming [Weinberg 1971], as well as information distribution methods such as program libraries [Luppino and Smith 1974], or the restriction of information availability [Parnas 1971].

These methodologies will be used increasingly in the future and will become an accepted aspect of programming efforts. Promising research is currently being done in areas such as program design based on data abstractions (see Section 4) and program construction based on more mathematically oriented problem solving [Dijkstra 1976].

2.4.1.2 Attention to Reliability After Implementation

After a program is constructed, its reliability can be enhanced by applying techniques of verification and validation, proofs of correctness, and measurement and modeling.

After software is written, it goes through a period of testing where an attempt is made to locate and correct mistakes. This is called verification and validation. "Verification is the process of determining whether the results of executing the software product in a test environ-

ment agree with the specifications," whereas "validation is the process of determining whether executing the system (i.e., software, hardware, user procedures, personnel) in a user environment causes any operational difficulties" [Smith 1975].

Verification and validation techniques can be classified as manual-based and computer-based. Manual-based methods include design verification, design validation, and code verification [Smith 1975]. Automated aids include program analyzers, testers, and simulators.

Basically the manual-based methods are readings or presentations of design documents or code and can be done by individuals or a specifically selected group of people. These techniques are an important aspect of verification and validation, and will see increasing usage in the future. In fact, they will overshadow the use of automated aids in the coming years. Partially this is because of the difficulty that automated aids are having at becoming accepted and used [Reifer 1975].

Program analyzers can be static or dynamic and are used to gather information about programs. Static information would include module connections, statement counts, and data usage. Dynamic information includes dynamic statement counts, values of variables such as minimum, maximum, first, and last, and branch counts [Stucki and Foshee 1975]. Some programs use the gathered information to generate test cases that will exercise the program being analyzed in an appropriate manner.

Testers try to analyze the workings of a program based on initial values, input values and assertions about the program's operation. This involves some sort of inequality solver. Initially this process was carried out with actual values. A recent innovation, which is very promising, is to carry out the execution symbolically. Two examples of symbolic executers are EFFIGY [King 1976] and SELECT [Boyer et. al. 1975].

In recent years research has been performed on automated aids both at universities and in industry. This is especially true for program analyzers and testers. Progress has been made in all areas and is expecte

to continue in the future. Tools exist right now that could be used profitably on real projects. Examples would be the PET system of McDonnell Douglas [Boettcher 1974], and the Jovial Automatic Verification System (JAVS) [Brooks et. al. 1976]. Testing tools are currently being investigated as an added task under this contract.

Despite these improvements, automated aids are having a difficult time becoming accepted. Some of the reasons for this problem are that 1) aids are often poorly structured, poorly documented, and poorly tested, 2) it is difficult to assess how cost-effective aids are, and 3) human factors such as poor communications and lack of management support are working against automated aids [Reifer 1975]. Lack of management support includes factors such as an unwillingness to devote employee and machine time to automated aids. For these reasons, automated verification and validation tools will not achieve widespread use in the next few years.

Ideally, one would prefer to prove that a program is correct rather than testing to achieve an acceptable level of reliability. Unfortunately, there are several problems with the "proof of correctness" approach. These include the additional effort required by the programmer and the limits of present day theorem provers. It will not be possible to use formal proofs of correctness in the near future.

Reliability measurement refers to the collection and classification of errors. This is an important task because results can be used to focus the design of reliability tools and techniques. It helps give an idea of what methods are needed and when they should be applied. One example was the use of error measurements to assess the error proneness of alternative language features in Gannon's study [Gannon and Horning 1975]. Another measurement project is described in [Shooman and Bolsky 1975].

Using the results of error measurement, one would like to set up error models. These models would be used to make statistical predictions about the reliability (level of errors) of programs. By placing an objective value on the reliability of a program, these reliability levels could

then be used to 1) control the testing phase of the project, and 2) evaluate the cost-effectiveness of various verification and validation methods.

Measurement is expected to proceed in the next five years leading to the gathering of much valuable data. Modelling and predictive methods are not as promising, however. Accepted results will probably not be obtained in the near future.

2.4.2 Compiler Reliability

As has already been mentioned, the main issues relating to compiler reliability are those that apply to general software reliability. In this section, issues relating specifically to compiler reliability will be discussed.

In compiler development, especially for a standard language, the design requirements are well defined. The language defining document constitutes an explicit specification of these requirements. Furthermore, methods for a complete formal language definition are possible through the use of semantic specification languages (see Section 3).

The next step in software development involves the logical decomposition or modularization of the program. Here again the problem is simplified when dealing with compiler implementation. A compiler is commonly divided into five distinct parts or phases: the scanner, parser, error recovery routines, optimizer and code generator. Each of these except the simplest, the scanner, has been discussed earlier in this section.

As the theory of each phase evolves, implementation becomes more reliable. This stems from the employment of general, well-understood algorithms. Traditionally the scanning and parsing phases have received the most attention. Several excellent algorithms for their implementation exist today. In the near future, general, consistent algorithms for

both optimization and error recovery will come into use. This will enhance the reliability of these sections of the compiler. Code generation is the least understood area of compiling and will remain so in the near future. For this reason, one can expect it to be the most likely source of reliability problems.

The evolution of general theories of compiling has enabled the development of automated aids for compiler writing. When used these tools increase the reliability of the resulting compiler. They eliminate much of the tedious work required for using new general algorithms. The tools range from systems which simply supply necessary compiler routines (e.g., dictionary procedures) to those which generate one or more complete compiler modules. Such tools will see increased use in the future, especially in conjunction with any large implementation effort such as that which would be required by adoption of a standard HOL.

The emergence of an oversight group in such a situation can certainly be predicted. Such a standards-enforcement agency could profit greatly through the use of compiler-writing tools. By utilizing these tools, the agency could supply sections of the compiler to implementation groups, thereby assuring adherence to the standard. For example, an agency-supplied parser would insure that a standard language syntax was accepted. Such modules, being developed only once and maintained by one group would increase the reliability of all compilers.

Traditionally compilers have been written in assembly language. The trend in this area, as in many others, has been away from this traditional approach and towards the use of high-order languages. Beside the usual advantages so gained, this makes it possible to write the compiler in the language it is meant to support. By doing this one can assure early and vigorous testing of the compiler. The major drawback to this approach is that a fairly large amount of code must be written before any testing takes place. This problem, of course, exists only for the first compiler for the language and can be minimized by the initial

implementation of a small language kernel. The methods employed for bootstrapping such a compiler are not uncommon.

The problems of proving compiler correctness are even more complex than those relating to general program correctness. What is required are three sets of axioms concerning three different domains: the source program, the compiler and the object code. Work in this area is not very promising for the near future. Examples of work of this nature can be found in [London 1971] and [Chirica and Martin 1975] .

While error measurement in relation to compiler development has yet to occur, this is potentially a very promising area. By limiting studies to this one very important area, one can hope to obtain more detailed and specific results.

In conclusion, the greatest future increases in compiler reliability will arise from the application of general well understood algorithms and the use of compiler-writing tools. A standards-enforcement agency is likely to emerge which will use compiler-writing tools to facilitate the maintenance of new language standards.

2.5 Summary

Over the next decade, there are four areas in which compiler technology can impact high-order language standardization. These areas are compilation techniques, compiler construction techniques, cross compilation, and reliability.

Recent advances have been made in compilation techniques that will see increasing use over the next decade. In the area of parsing, LALR and LL parsing methods will be used because of the ease with which they can be automated. The automatic generation of parsers will both increase their reliability and simplify the problems of language standardization. More accurate error detection coupled with systematic error recovery methods will see increasing use. Finally, global optimization methods, being of a machine-independent nature, will be more widely used.

Compiler construction includes automated and non-automated methods. Non-automated methods have and will continue to benefit from techniques such as structured programming that improve general programming practices. Use of automated methods such as compiler writing systems will continue to increase, especially in the areas of parsing and error-handling.

Considering the number of compilers that would need to be written for any new language agreed upon as a standard, automated compiler construction aids are likely to be utilized. In addition, some agency will most probably emerge with the responsibility for maintaining and enforcing the language standards. The work of such an agency would be greatly facilitated if they could supply to implementors automatically generated sections of the compiler. These sections may vary in complexity from simple parsing tables to the compiler's entire front-end (i.e., the scanner, parser, error-handling routines and machine-independent optimizer). An example of this type of work is the JOCIT system for JOVIAL/J3.

While it would be convenient to be able to produce an entire compiler, advances in the area of code generation are much slower in coming. Another area needing further investigation is intermediate language standardization. This could facilitate compiler construction and greatly reduce the cost.

It should be remembered, however, that the cost of compiler construction is often controlled by external factors such as (1) constraints on the size and performance of the compiler, (2) constraints on the size and performance of object code produced by the compiler, (3) project deadlines, and (4) host and target machine characteristics. The selection of host and target machines is typically determined by factors other than their impact on compiler development costs.

Cross compilation is an important technique that will continue to be used in the future. It could greatly benefit from a standard intermediate language since the replacement of the machine-dependent portion of the compiler would be greatly facilitated.

Increased attention to reliability will occur both during and after implementation. Tools and techniques for building reliability into compilers include formal specification languages, properly designed programming languages, program design techniques, program structuring techniques, and project management techniques. All of these techniques except formal specification languages have been used and will see increasing use in the future. Techniques that can be used for enhancing reliability of existing software are verification and validation techniques, proofs of correctness, and measurement and modelling. Good testing tools exist for verification and validation, and they will continue to improve in the future. However, they will not be widely used unless official policies change. There will be advances in the area of error measurement but proofs of correctness and modelling will not be used in the near future.

3.0 SEMANTIC SPECIFICATION TECHNOLOGY

3.1 Introduction

The use of programming languages requires a method for communicating the meaning of those languages to persons involved with their design, implementation, and use. The large number of proposed methods indicates a lack of consensus on how the meaning of programming languages is most effectively communicated. The current situation is well described in the Introduction of [Marcotty et. al. 1976]:

> The programming language Tower of Babel is well known. Less discussed is the Tower of Metababel, symbolic of the many ways that programming languages are described and defined. The methods used range all the way from natural language to the ultra-mathematical. The former are subject to all the vagaries and inconsistencies that result from normal prose, and the latter frequently have their meaning hidden under abstruse notation.

The development of Backus-Naur Form (BNF) was a major advance in the formal specification of (context-free) syntax for high-order languages. BNF and its notational variations have been almost universally adopted as a syntactic specification technique. To some extent it is the success of BNF that has strengthened the desire for a similarly-useful technique for semantic specification. (Unless noted otherwise, we will assume the definition of context-sensitivities is included in a "semantic" specification.)

A major concern for semantic specification is to develop or select a technique with the following characteristics:

1) Completeness. The technique must ensure that the semantics are completely specified. When syntactic and semantic specifications are coupled to form a language definition, they must resolve all questions about the language.

2) Preciseness. The technique must also ensure that the specification be precise and unambiguous, in order to avoid misinterpretations of the intent of the definition.

3) Understandability. The technique must utilize a notation which is simple, natural to the users of the definition, and learned with a minimum of effort. The technique should encourage the creation of specifications that are clear and understandable.

4) Pragmatism. The technique must provide a means for encapsulating those portions of the semantics that are dependent upon the implementation environment. Constructed in this way, a definition could be used by a language standardization group in deciding if a particular implementation conforms to the "standard" semantics of the language.

In the following subsections, we briefly discuss the role of semantic specification in language standardization and control. Conceptual approaches to semantic specification are considered, and currently-existing techniques are surveyed. An analysis of these techniques and on-going research, as described in the technical literature and by personal contacts, forms the basis of a technology forecast for semantic specification techniques. Finally, we present our conclusions about the relationship between forecasted developments and language standardization activities.

3.2 Role of Formal Specification in Language Standardization.

The role of formal specification should be considered in the broadest view of language standardization activities. These include the initial design or selection of the language, the maintenance of the language standard as changes are made to correct flaws or to enhance the design, the construction and testing of actual implementations, and most importantly, the precise communication of the standard to those involved in its use.

Several examples of the use of formal languages in the above areas

can be cited. Mr. Paul Berning, of the SEMANOL specification language group at TRW, has cited the value of maintaining a formal definition for the language UCMS-2 concurrently with the language's design [Berning 1976]. A similar opinion was expressed in [Ledgard 1976], who observed the usefulness of formal language techniques in revealing complexities and design flaws during language development at the University of Massachusetts. There is not as much experience in using formal definitions as the fundamental basis for maintaining a language standard or for evaluating actual implementations. However, the recently-accepted standard for PL/I is defined in [BSR X3.53 1975] using a specification technique very similar to the formalism of the Vienna Definition Language (described later in this report); this will be the first real test of using a formal definition to maintain a widely-used standard.

Some of the consequences of not using a formal definition are cited in [Marcotty et. al. 1976]:

> Computer science has already made considerable progress without having a generally accepted formal technique for defining programming languages, just as the English language was well developed before the advent of Johnson's Dictionary of the English Language in 1755. However, the lack of general use of formal definitions has not been without severe consequences. For example:
>
> a) There is still confusion over the difference between syntax and semantics.
>
> b) Standardization efforts have been impeded by a lack of an adequate formal notation.
>
> c) Despite the fact that there exist standards for programming languages, it is still chancy to move a program from one implementation to another, even on the same hardware.
>
> d) It is impossible to make a contract with a vendor for a compiler and be assured that the product will be an exact implementation of the language.
>
> e) It is difficult to write reference manuals and tutorial texts without glossing over critical details that may change from implementation to implementation.

> f) The answers to detailed questions about a programming language frequently have to be obtained by trying an implementation or hoping for a consensus from several implementations.
>
> Most of these problems would be avoided if there were good formal definitions for the languages.

Indeed, the need for a useful specification technique to overcome problems related to language standardization has long been recognized. For example, the following statement was made almost a decade ago:

> From a purely scientific viewpoint, the members of the various working groups concerned with programming language standardization really ought to report to their parent committees that their assigned task is impossible without a major prior effort by the technical community; and that this prior effort would have to produce an effective procedure for describing the languages that are of concern. [Steel 1967]

An important point in Steel's statement is that a specification technique be "effective." That is, in addition to being formal (i.e., complete and precise), a technique must be understandable to, and accepted by, the technical community. We will now consider the various approaches and techniques for semantic specification, particularly with regard to their effectiveness.

3.3 Approaches to Semantic Specification

There are two principal approaches to semantic specification; a given specification technique will be either "compiler-oriented" or "interpreter-oriented." A compiler-oriented technique specifies the meaning of a programming language by defining how any program in the language can be translated into another language whose semantics are known. An interpreter-oriented technique defines, for each program in the language and any legal input data, an algorithm for computing the result of executing the program with that data. Techniques from either approach are generally "syntax-based" in that translation or execution semantics are associated with

syntactically-distinguished program components. [Wegner 1972] summarizes the characteristic differences between compiler-oriented and interpreter-oriented specification techniques:

> A compiler-oriented language definition associates with each production of the form $X \rightarrow Y_1Y_2...Y_k$ a compile-time semantic action $f_x(Y_1, Y_2, ...,Y_k)$ that may include the generation of target language code and the updating of compile-time state variables used in controlling the compilation process. ...An interpreter-oriented language definition associates with each production $X \rightarrow Y_1Y_2...Y_k$ a state transformation from the current state I_j to a new state I_{j+1}.
>
> In a compiler-oriented language definition, the recognition that a substring of a program is generated by a given production gives rise to a translation step that may generate a target-language string... In an interpreter-oriented language definition the recognition...of a specific syntactic structure gives rise to the execution of a sequence of instructions... Moreover, a compiler requires a given source-language string to be scanned at most once for each pass of the compiler, while an interpreter may require a given source-language string to be executed an arbitrarily large number of times.
>
> A compiler-oriented language definition avoids the necessity of specifying execution-time semantics by assuming that the execution-time semantics of the target language is given as a primitive in terms of which source languages may be defined. This allows many semantic attributes of compiler-oriented definitions to be characterized by compile-time algorithms that are guaranteed to terminate. However,...solving a problem A by reducing it to the solution of a previously solved problem B...assumes that the problem B has been solved. ...Thus, a compiler-oriented definition merely postpones the difficult problem of defining the programming language interpreter without eliminating it.

In [Donahue 1975b], the interpreter-oriented techniques have been classified further into three categories: operational, denotational, and propositional.

> An operational model of a programming language is given by 1) defining an abstract "machine state," S, containing the essential information about the progress of the computation invoked by each program in the language, and 2) specifying the meaning of constructs in the language as their effect on the state, i.e., by a state transition function....
>
> The approach taken in denotational definitions is to abstract the operational view of meaning and to consider the program as specifying a function of some appropriate type. Thus, meaning is given not in terms of state sequences, but simply as functions from states to states... By giving a purely functional description of the language, the explicit sequence of operations given in the operational definition is now [in the denotational definition] only implicitly specified in the definition of functional composition.

Thus the denotational approach abstracts the essential properties of the operational approach, while removing the notion of state sequences. The third approach, using propositional techniques, furthers the process of abstraction by eliminating the notion of states themselves. As described by Donahue:

> The basic set of objects in the propositional approach, rather than a set of "states", is the set of formulas of some logical system.

The theoretical aspects of the various approaches to semantic specification have been studied by many persons. Different techniques for associating compiler-oriented translation steps with syntactic constructs were proposed in [Irons 1961, Brooker and Morris 1962, Irons 1963, Wirth and Weber 1966, and Feldman 1966]; the theoretical properties underlying these techniques were studied in [Lewis and Stearns 1968]. More recent compiler-oriented techniques have been proposed in [van Wijngaarden 1966, Knuth 1968, and Ledgard 1969]. Operationally-modelled interpreter-oriented techniques are best illustrated by the lambda-calculus work of [Landin 19].

The compiler and operational interpreter approaches have been combined in the work of [Landin 1966], the IBM Vienna group [Lucas et. al. 1968], and the TRW SEMANOL group [Blum 1969a, 1969b, and 1970]. Under these combined schemes, a translation is defined from source language programs into a more abstract semantic basis (e.g., a lambda-calculus expression or an abstract parse-tree structure); the definition is completed by specifying an interpreter which gives meaning to the semantic basis in terms of an execution model.

Examples of denotational approaches are provided in the work of [McCarthy 1960, 1963a, and 1963b] and [Scott and Strachey 1971]. Propositional approaches are fundamental to the inductive assertion technique of [Floyd 1967] and the axiomatic method described in [Hoare 1969a and 1971a]. More recently, [Hoare and Lauer 1974] and [Donahue 1975b] have investigated the possibility of using multiple approaches to provide complementary sets of language definitions. Their intent is to provide compatible definitions at differing levels of semantic abstraction, so that definitions at differing levels can be oriented towards different audiences. (For example, a language implementor might wish to view a language via an operational approach while someone concerned with the correctness of programs in the language might prefer a propositional approach.)

3.4 Existing Techniques for Semantic Specification

In the preceding subsection, the theoretical approaches to semantic specification were overviewed briefly. We now consider the important techniques that have been developed using those approaches.

3.4.1 Compilers

An early suggestion for a specification technique was to have programming languages be defined by their compilers. [Garwick 1966] begins by stating that

> No programming language for a given computer can be better defined than by its compiler.

However, he goes on to observe that

> If the proposal to define languages by their compilers is to be at all possible, there are two conditions which must be fulfilled. The compilers must be machine independent, and they must be readable.

Garwick proposed that the compilers be made independent by writing them in a machine-independent language (he proposed ALGOL 60) and having them generate code for a "machine independent computer." The MIC would be either an abstract or a real machine, with an assembly language similar to that of "any reasonably conventional machine having instructions that are not too special."

In response to Garwick, Hoare noted that:

> Every implementation of a programming language is, I think, a precise definition of the semantics -- not necessarily an accurate one incidentally. Any absolutely precise definition of a language -- semantic definition -- is (or can be) regarded as an implementation. If we want to define a language which is capable of more than one implementation (which for many reasons we do) we must avoid, or be very suspicious of, any attempt to give an absolutely precise semantic definition of the language. What is required is a method of describing a class of implementation perhaps by giving a criterion for testing whether the implementation is satisfactory....First, we must give a great deal of thought to deciding which things we want to leave imprecisely defined. Second, in any formal or informal description of a language, we must have a mechanism for failing to define things. [in Steel 1966, pp. 142-143]

A very successful example of the compiler technique is the definition of the EULER language in [Wirth and Weber 1966]. An algorithm is given for translating EULER programs into code for an abstract EULER machine; also included in the definition is an algorithm for interpreting this object code. An example of the impreciseness mentioned by Hoare is that storage "cells" in the EULER machine are allowed to contain lists of unrestricted length (this was possible because the definition existed on paper). The flexibility of the definition was demonstrated by modifications made in support of parallelism [Chirica et. al. 1973].

However, it is generally agreed that machine dependencies were not eliminated completely from the EULER definition. In addition, the readability of the definition is as much due to the generality and conceptual simplicity of the language as it is to the specification technique. It is doubtful if a similar technique could be applied as successfully to more sophisticated (some would say unnecessarily complex) programming languages. The next two techniques we consider have attempted to provide greater readability by defining the compilation process, both functionally and notationally, at a higher level of abstraction.

3.4.2 Production Systems

This specification technique has its theoretical roots in the works of [Post 1943, Smullyan 1961, and Donovan and Ledgard 1967]. The current technique has evolved through the more recent work of [Ledgard 1969, 1972, 1974, and 1975]. A production system definition contains a context-free grammar in a form somewhat like BNF. Auxiliary functions are defined to manipulate n-tuples whose primitive components are taken from the underlying semantic base (e.g., Integer and Boolean values). Functions associated with individual productions in the grammar use the auxiliary functions to define sets of n-tuples.

When given the productions associated with a particular program, the functions define a set of n-tuples representing the "meaning" of the program in terms of the primitives. N-tuples representing "syntactic environments" are defined and used to specify the context-sensitive requirements of the language. In a similar fashion, a mapping is defined from each syntactically-legal program into a target language. An example definition in [Marcotty et. al. 1976] maps legal ASPLE programs into proof verification rules. The meaning of the rules is based on underlying assertions according to Hoare's axiomatic approach (discussed later in this section).

Production systems can certainly be used to define translations into other target languages in addition to an axiomatic basis. Ledgard also states that this technique could be used to define semantics directly (i.e., follow a more interpreter-oriented approach). A set of triples could be defined, specifying the output corresponding to each syntactically legal program and its allowable sets of input. This approach has not yet been attempted with production systems.

3.4.3 Attribute Grammars

The specification technique of attribute grammars, also known as declarative semantic definition, was originally proposed in [Knuth 1968]. Related concepts have since been described in [Koster 1971] and [Lewis et. al. 1973]. An attribute grammar consists of a context-free grammar (usually described using a BNF variation) where each syntactic entity has an associated set of attributes. Associated with each production in the grammar is a set of functions that define the values for attributes of entities occurring in that production. The types of attribute values are taken from the underlying semantic base.

For a given program, the context-free grammar defines a derivation (or parse) tree whose nodes correspond to the syntactic entities. Thus, each node has its own set of attributes. The value of an attribute may either be inherited (i.e., functionally dependent upon attribute values associated with the immediate parent of the node) or synthesized (i.e., functionally dependent upon attribute values associated with immediate descendents of the node). Chains of attribute value dependencies are terminated when values are taken directly from the primitives in the semantic base. The meaning of the program is given by the synthesized value of an attribute associated with the root node of the tree. Typically, this value is (a character string representation of) a sequence of statements in a target language, forming a translated version of the program.

The practicality of this technique was first demonstrated by [Wilner 1971], who used it to define the semantics of SIMULA 67 in terms of an instruction set for a sophisticated abstract machine. Wilner was able to simplify the definition by eliminating "global" attributes. He also designed a set of synthesized and inherited "environment attributes" which are used to specify context-sensitivities in a fashion similar to the "syntactic environments" of the production systems technique.

A declarative semantic definition was developed by [Dreisbach 1972] which specified the semantics of PL360 in terms of the IBM 360 machine instruction set. [Berry 1974] used the technique to define the language OREGANO in terms of an abstract VDL tree. (The semantics of the tree structure were defined by a VDL interpreter, as will be discussed in the section on the Vienna Definition Language.) A system for evaluating attribute values automatically was proposed in [Fang 1972], and recent research into techniques for more efficient evaluations have been described in [Bochmann 1974 and 1976a].

3.4.4 W-Grammars

This technique was first used to define the language ALGOL 68 in [van Wijngaarden et. al. 1969, 1973, and 1975], although some of its underlying concepts were proposed earlier in [van Wijngaarden 1966]. Definitions using this technique are composed of two related grammars. Production rules in the first grammar, termed hyper-rules, act as templates for context-free productions. Production rules in the second grammar, termed meta-productions, are used to generate terminal strings from starting entities in the hyper-rule templates. Since the meta-productions, which are also context-free, can produce an infinite set of terminal strings, the replacement of these strings for their starting entities in a given hyper-rule can potentially result in an infinite number of context-free productions.

The syntax of the language is defined by taking the union of all of the productions which can be created from the hyper-rules. Because there can be an infinite number of productions, the technique can be used to define not only context-sensitivities, but also the language semantics. In a manner too detailed to be described here, "snapshots" of the state of execution can be made a part of the syntactic entities. Thus a legal program will have a W-grammar derivation tree whose terminals consist of the program, the initial state of a legal input file (more than one of which can be derived for each program), the final state of a corresponding output file, and nothing else. The "nothing else" portion is important in that constraints in the definition are specified by having non-terminal strings, for example "where typeof alpha equals typeof beta", reduce to the empty string only when the constraint is properly met. A variation on the technique is to use a W-grammar for a compiler-oriented approach in which derivation trees would define both source programs and their corresponding translations into the target language.

The major drawback of the W-grammar technique is that, although it has a conceptual simplicity, its application to the definition of languages can be quite complex. This complexity significantly impacts on the effectiveness of the technique, since definitions can only be understood and used by persons who are intimately aware of the notation and other conventions. An obvious example of this problem is given by the difficulties generally encountered when reading the ALGOL 68 reports.

3.4.5 Vienna Definition Language

This specification technique, referred to as VDL and sometimes UDL (for universal), was developed by the IBM language definition group in Vienna [Lucas et. al. 1968]. It was used by that group to define both ALGOL 0 [Lauer 1968] and PL/I [IBM 1966, Lucas and Walk 1969, and Bekic et. al. 1974]. Understandable descriptions of the technique are provided in

[Neuhold 1971 and Wegner 1972], although the most concise yet understandable overview is provided in [Marcotty et. al. 1976]:

> A formal definition in VDL is based on the concept of an abstract machine. ...The meaning of a program is defined by a sequence of changes in the state of the abstract machine as the program is executed. The rules of execution are defined by an algorithm, the Interpreter. To make a distinction between those properties of a program that can be determined statically and those that are intrinsically connected with the dynamics of the program's execution, the original program is transformed into an abstracted form before execution. This transformation is performed by another algorithm, the Translator... During the transformation, the context-sensitive requirements of the syntax can be checked.
>
> In VDL both the abstract machine and the program are objects. An object can be represented as a tree. There are two classes of objects: elementary objects, with no components, and composite objects, with a finite number of immediate components that are also objects. Thus, in a tree representation, an elementary object is a terminal node and a composite object is a nonterminal or branch node.

The Vienna Definition Language is simply a notational technique for defining these trees. The important point is that the same notation can be used to describe trees which represent the program, its data, its output, the Translator functions, and both the storage and control states of the Interpreter during its execution.

In a formal definition, the notation is used to specify a tree representation for each program in the language based on its context-free syntax. The Translator is a composite function, which, given the initial tree for a program, defines the construction of another tree that represents the program without syntactic "sugaring" but with resolved context-sensitivities. Depending upon the resulting "abstract" program tree, an input data tree, and the current state of a "storage" tree in the Interpreter, functions in a "control" tree (also in the Interpreter) define changes to an output tree, the storage tree, and the control tree itself. Thus given an initial state of input, control, and storage for a program, the Interpreter functions define a series of

state-to-state transitions which ultimately result in the final output for the program.

Use of the VDL technique has been widespread. It was used to define APL in [Gerhart 1971] and ALGOL 68 in [Uzgalis et. al. 1973 and Kelly 1974]. An interesting application was made in [Berry 1971] to define the semantics of the Contour Model for block-structured processes [Johnston 1970].

Although the notation greatly unifies the various aspects of a VDL definition, its mathematical style has not always been well liked. Recent attempts have been made to use a VDL approach with a more natural-language notation, as in the definition of the recently-accepted PL/I standard [BSR X3.53 1975]. The effectiveness of this approach will be determined as that definition becomes more widely distributed and used.

3.4.6 SEMANOL

In a manner similar to VDL, SEMANOL (Semantic-Oriented Language) defines programming languages through a compiler/operational-interpreter approach. Original development of the technique is presented in [Blum 1969a, 1969b, and 1970]. Further development of the technique was done at TRW for the U. S. Air Force, as reflected in [TRW 1973b, Blum 1973, and Berning 1973, 1975b and 1975d]. The technique has been used to define the JOVIAL (J3) and (J73) languages [TRW 1973a and Berning 1975a and 1975c] and similar efforts have been reported for BASIC and UCMS-2 (a U.S. Navy language).

Comparisons have been made between SEMANOL and VDL in [Anderson et. al. 1974 and 1976]. A major difference is that the SEMANOL notation is in the form of a programming language; thus a formal specification for a given language is defined by a SEMANOL program. (This technique permits the syntax and semantics of the SEMANOL language to be defined in terms of itself.) The programming language notation allows the construction of a working interpreter which can test a formal definition

using sample programs from the language being defined. It is also claimed that definitions written in the SEMANOL language are more readable (i.e., understandable) than definitions with other, non-language notations.

> A SEMANOL program, p_L say, specifies the syntax and semantics of some programming language L. For an arbitrary program p in L, the metaprogram p_L specifies: (1) the output produced when p is executed on some given input data and (2) an algorithm which p prescribes for the computation of this output. Thus p_L gives both (1) the extensional [i.e., denotational] meaning of p as a function f mapping inputs onto outputs and (2) the intensional [i.e., operational] meaning of p as an algorithm for computing values of f. The SEMANOL program p_L actually emphasizes (2), since p_L is itself a program which, when executed, interprets p as an algorithm in terms of the SEMANOL system's basis operations and relations.
>
> A *SEMANOL programmer* will usually begin writing a program [i.e., a language definition], p_L, by writing a context-free grammar which specifies an underlying syntactic structure of the object language L. ...Thus, p_L contains a set of "syntactic definitions" which look very much like a BNF syntactic description of L. ...The non-context-free syntax and the semantics of L are prescribed by a set of "semantic definitions."...The semantics of L is given in definitions which specify evaluation, that is, how values are to be assigned to formulas in a program p, and control, that is, the order of execution of the "statements" of p. ...These definitions are SEMANOL statements of a declarative nature. To make a SEMANOL program, p_L, executable there are also elementary command statements (e.g., simple assignment)... Execution of p_L can actually be carried out by using an interpretive program, M, now operational on the HIS 6180 computer. M will accept p_L and any object program p in L together with input data E_1 for p. M will cause p_L to execute interpretively

on (p,E_1); i.e., p_L invokes execution of program p on the data E_1 [Anderson et. al. 1976].

In the section forecasting future developments for semantic specification technology, we address some of the current problems with the SEMANOL technique as well as planned directions for further development.

3.4.7 Lambda Calculus

The lambda calculus is a language originally proposed in [Church 1941] for the specification of functions. Its syntax (there are actually several variations) and semantics are so remarkably simple and concise that we present them here (in BNF and English):

Syntax:
```
<lambda-exp> ::= <identifier>
               | λ <identifier> <lambda-exp>
               | <lambda-exp> ( <lambda-exp> )
```

Definitions: Identifiers will be called variables. A binding variable is a variable immediately following a λ symbol. A bound occurrence of a variable x within a lambda expression M is either an occurrence of x as a binding variable, or an occurrence of x within a lambda expression N where λxN is an expression in M. A free occurrence of x in M is any occurrence of x not bound in M.

The notation $S_A^x M$ represents a lambda expression resulting from the substitution of expression A for all occurrences of variable x in expression M.

Semantics:

1. Reduction: An expression $\lambda xM(A)$ can be replaced by $S_A^x M$ provided M contains no bound occurrences of x and A contains no free occurrences of variables that are bound in M.
2. Renaming: If M is an expression other than λx containing bound occurrences of x, then M can be replaced by $S_y^x M$ provided M contains no free occurrences of x and no occurrences of y.

The above definition is a modified version of one presented in [Wegner 1968]. The expression $\lambda x M(A)$ can be thought of as a function, where M represents the body, x the formal parameter, and A the actual input value. The meaning of the function is specified by the reduced expression resulting from the application of the reduction and renaming semantics.

The application of this technique to programming languages is due to [Landin 1964], who described an "SECD" machine to interpret lambda expressions and produce their corresponding reduced form. In [Landin 1965 and 1966], a definition of ALGOL 60 was developed by specifying a translation from ALGOL 60 programs to lambda expressions and then giving those expressions interpretive meanings in terms of the SECD machine. However [McCarthy 1966], referring to compiler-oriented definitions in general but specifically including the Landin technique, argued that:

> These definitions have a certain practical value in resolving ambiguities, but they do not correspond to our intuitive ideas; they will make mathematical results difficult to obtain....

This discussion of lambda calculus has been included because of its historical significance to both the defining and the understanding of the semantics of programming languages. However, as a practical tool for semantic specification, lambda calculus has been superseded by more recent techniques.

3.4.8 Mathematical Semantics

Mathematical semantics is not a specific technique for defining programming languages; instead, it is a philosophy that programming language semantics should be considered from a mathematical (i.e., functional or relational) point of view rather than from an algorithmic one. Thus mathematical semantics is based on the denotational (and to some extent propositional) approach as opposed to the operational approach.

> The mathematical approach rejects the view that semantics can be given by an algorithm for getting the result of program and data together, as in operational definitions. Instead, the mathematical approach defines semantics by assigning meanings of whatever type may be appropriate (functions or relations of some kind) directly to programs and program constituents. The Scott theory [see references below] is an attempt to provide the abstractions necessary to give such a semantics.
>
> The theory is basically an answer to two fundamental questions:
>
> 1. What is a data type? The theory defines data types as complete lattices.
> 2. What characterizes the class of permissible mappings between data types? The theory assumes that mappings between data types are continuous (mappings preserve limits).
>
> [Donahue 1975a].

The original "mathematical theory of computation" was proposed in [McCarthy 1960, 1963a, and 1963b], although a large portion of the theoretical development is due to [Strachey 1966, Scott 1970, Scott and Strachey 1971, Strachey 1973, and Strachey and Wadsworth 1973]. This work has been followed more recently by that of [Tennent 1973 and 1975]. Because of the mathematical formalisms in this approach, it has not yet been accepted widely as a practical definitional technique by the "programming community." However, research studies are continuing, and there exists the potential for developing more practical specification tools based upon the theory.

3.4.9 Axiomatic Approach

This technique began with the concern for program verification [Floyd 1967], which of course is highly dependent upon an accurate understanding of the program's semantics. The approach was developed in the work of [Hoare 1969a, 1971a, 1971b, 1972a, and 1972b] and used to define the language PASCAL in [Hoare 1969b] and [Hoare and Wirth 1973].

> To define semantics using an axiomatic approach, the following question is addressed: upon termination of a program, what assertions can be made? The axiomatic approach . . . is based on the first-order predicate calculus [Manna 1969] which permits assertions about the membership of objects in sets and the results of applying operations to objects, e.g., the kinds of objects stored on some external medium and the values of expressions. To define the semantics of "programs", a correspondence between programs and the relevant assertions must be defined. [Marcotty et. al. 1976].

> In the Hoare system, statements of the programming language are identified with relations between assertions, where
>
> 1. atomic statements are characterized by axioms or axiom schemas, and
> 2. compound statements are characterized by rules of inference with one or more premisses. [Donahue 1975b]

A strong justification for using the axiomatic approach for defining language standards was made in Hoare's original paper [1969a]:

> Apart from giving an immediate and possibly even provable criterion for the correctness of an implementation, the axiomatic technique for the definition of programming language semantics appears to be like the formal syntax of the ALGOL 60 report, in that it is sufficiently simple to be understood both by the implementor and by the reasonably sophisticated user of the language. It is only by bridging this widening communication gap in a single document (perhaps even provably consistent) that the maximum advantage can be obtained from a formal language definition.
>
> Another of the great advantages of using an axiomatic approach is that axioms offer a simple and flexible technique for leaving certain aspects of a language undefined, for example, ranges of integers, accuracy of floating point, and choice of overflow technique. This is absolutely essential for standardization purposes, since otherwise the language will be impossible to implement efficiently on differing hardware designs. Thus a programming language standard should consist of a set of axioms describing the range of choices facing an implementor. ...
>
> Another of the objectives of formal language definition is to assist in the design of better programming languages. The regularity, clarity, and ease of implementation of the ALGOL 60 syntax may at least in part be due to the use of an elegant formal technique for its definition. The use of axioms

> may lead to similar advantages in the area of "semantics," since it seems likely that a language which can be described by a few "self-evident" axioms from which proofs will be relatively easy to construct will be preferable to a language with many obscure axioms which are difficult to apply in proofs. Furthermore, axioms enable the language designer to express his general intentions quite simply and directly, without the mass of detail which usually accompanies algorithmic descriptions. Finally, axioms can be formulated in a manner largely independent of each other, so that the designer can work freely on one axiom or group of axioms without fear of unexpected interaction effects with other parts of the language.

3.4.10 LCF

LCF is a logical calculus whose formal properties are described in [Milner 1972a and 1972b]. The technique has been used to describe the semantics of PASCAL [Aiello et. al. 1974]. In addition, a proof-checker has been implemented which is claimed to be capable of interpreting LCF definitions of programming languages [Aiello and Weyhrauch 1974]. However, a more thorough investigation of the technique was beyond the scope of this report.

3.5 Forecasted Developments in Semantic Specification Technology

The following forecast is based upon our analysis of the current state-of-the-art and on-going research in specification techniques. Also considered were the comments and opinions received through personal communications with Paul Berning and E. K. Blum of TRW, Gregor Bochmann of the University of Montreal, Captain John Ives of the U. S. A. F. (Rome Air Development Center), Henry Ledgard of the University of Massachusetts, and J. Allen Robinson of Syracuse University.

3.5.1 General Forecasts

1) The value of using formal definitions as an integral part of programming language design will become more widely recognized.

Specification techniques will be used to aid the analysis of both overall language design and particular language constructs.

2) The value of formal definitions as a configuration control tool for language standards will also be more widely recognized. However, their actual usage will be limited over the next few years as research continues on ways to make the techniques more acceptable (i.e., understandable and usable) to the general "programming community". Formal techniques can not only be used to maintain a standard language definition, but also have the potential for use in automatic test case generation for the certification of standard compilers. However, this is an area that has seen little research activity in the past. Advances in the state-of-the-art are required before accurate forecasts can be made about its practicality.

> 3) The interface with users is the key area where most of the effort is needed. The meta-language of a formal definition must not become a language known to only the high priests of the cult. Tempering science with magic is a sure way to return to the Dark Ages. [Marcotty et. al. 1976]

The general emphasis in future research will be on how to make effective use of the existing techniques (although this may result in the development of improved variations). There will be a concentration on finding standard approaches to their use, rather than investigations into entirely new techniques. More research is also needed into approaches for explicitly defining program errors and for indicating places where implementation-dependent variations can occur.

4) There is not currently a general agreement as to which conceptual approach or specific technique is the best. The trend in the academic/research community is toward the more mathematical (i.e., denotational and propositional) approaches, in the belief that these follow more closely our intuitive understanding of language semantics.

The trend away from operational approaches is in keeping with the emphasis on abstraction in current programming language design; it is an attempt to describe problems (including that of language definition) at the user's level rather than at a machine level (even as reflected in an abstract execution model).

5) Although the emphasis will be on mathematical definitions for semantic concepts, there is still a future for using the compiler-oriented techniques of production systems and/or attribute grammars. Their value is that they provide a straightforward means for associating underlying abstract concepts with specific features (and frills) of any given language. While proponents of the two techniques disagree as to which contributes more to understandable definitions, they generally agree that the attribute grammar technique has much more potential for direct carry-over into language implementations, in the same sense that models of grammar (e.g., LALR) were carried over in the construction of syntactic parsers.

6) A promising goal for the future is to define languages at the source level in terms of a simple semantic base, using a declarative semantic (i.e., attribute grammar) technique. A definition would be completed by defining the semantic base via a set of axioms as suggested by Hoare. Users of the language would primarily need to understand only the declarative semantics and could be provided with an intuitive natural-language description of the semantic base. More sophisticated designers and implementors of the language would also be concerned with the formal description of the underlying semantic primitives given by the axioms.

One drawback to the proposed technique is that, since internal state-to-state transitions are not defined, it might not be the best technique for judging the correctness of actual implementations. One suggestion is to develop an operational-interpreter definition of a particular implementation, or class of implementations, which could

be checked against the language definition for overall semantic correctness; actual implementations in that class could then be checked against the execution model interpreter.

Although the above is a promising goal, we do not feel it will be realized in the near future, since its success depends upon the research described in paragraphs #2 and #3. For the short term, language standardization and control will rely more on the operational approaches of VDL (as in the PL/I standard) and SEMANOL.

3.5.2 SEMANOL Forecasts

1) SEMANOL is not as widely known or accepted in the academic/research community, and therefore probably has not been scrutinized as closely as the other techniques. Hopefully it will be given a closer look in the future.

2) We are not aware of a widespread use of the SEMANOL specification of JOVIAL (J73) among users of that language and thus are unable to comment on the reactions of the programming community.

3) The developers of SEMANOL have claimed several advantages over VDL:

a) The programming language notation makes SEMANOL definitions more readable.

Based upon our brief investigation, we believe the opposite is true; the syntactic notation in the language is not concise, not clear, and in fact less readable than most programming languages or functional notations. Similar opinions were expressed by others contacted during our investigation. The developers have indicated an awareness of this problem, and are hoping to correct it in a revision called SEMANOL 76.

b) The fact that SEMANOL definitions are executable programs provides the ability to determine the intent of subtle semantic issues through actual interpretation.

We agree with the basic concept, but are not convinced it can be applied in a practical sense. It is our understanding that only the syntax and simple portions of the semantics of JOVIAL have been tested; if possible, rigorous testing of the non-trivial semantics is required to resolve this issue.

c) More care will be taken by the definers of a language if they are aware the definition will be subjected to actual testing.

We have found a similar effect during the writing of sample programs (as opposed to operational ones).

d) SEMANOL utilizes the concept of a parameterized family of language definitions, where appropriate parameters can be used to specify machine-dependent language restrictions.

Certainly this is an important ingredient in a language definition.

4) During our investigations, contacted persons raised several other questions about SEMANOL:

a) Does the technique lend itself to reducing complexity in a definition?

Just as in other operational-interpreter approaches such as VDL, the SEMANOL technique may or may not eliminate a certain degree of complexity; however, it does not permit a more abstract (and therefore simpler) view of the semantics as can be done with denotational and propositional interpreter approaches, and to a lesser extent, with the functional notation of the declarative semantic compiler approach.

b) Can the SEMANOL method be used to define languages with parallelism, and thus non-deterministic behavior?

The SEMANOL design is highly-dependent upon a "control loop" for the interpretation of statement sequences. It does not have the equivalent of the VDL "choice" function, which could be used to describe non-determinism. Again, the developers of SEMANOL

have indicated an awareness of this problem, and are considering various solutions. We are concerned that such solutions, in altering a fundamental part of the SEMANOL design, might be ad hoc at best. In addition, modelling non-determinism will be difficult given the constraint that the definitions be truly executable.

c) What basis should be used in choosing the primitive relations and operational functions of the SEMANOL system?

The choice of system primitives is an area of current research in the SEMANOL group. We wonder how they can be chosen to avoid dependencies due to the implementation environment for the system, which might be carried over into the language definitions.

5) The Air Force has expended a great deal of effort on the SEMANOL development. As further insight into semantic specifications is gained, enhancements will be made to improve the SEMANOL system, just as we have seen enhancements to FORTRAN based upon insights into structured programming. Currently planned improvements and activities include:

a) the use of SEMANOL as part of the Language Control Facility being established at the Rome Air Development Center. A part of this effort will be devoted to modifying the system to aid in compiler verification and hence language control. These modifications will include automatic test case generation from a formal definition. Although compiler verification is an admirable goal, it is also an extremely difficult task.

b) a compiler mode for the implementation whereby SEMANOL programs could be compiled into an intermediate form and thus improve the efficiency of interpreting sample programs from the defined language.

The developers have indicated other areas of possible long-range activity, including

a) the development of a compiler-compiler mode, whereby the system could aid in the direct generation of compilers for the language being defined,

b) the investigation of optimization techniques, and

c) using the system as a basis for studies into automatic program verification.

Success in any of these areas, while not likely in the near future, will have a large and positive impact on language standardization activities.

4.0 HIGH-ORDER LANGUAGE TECHNOLOGY

This section identifies some major trends in current HOL technology, describes what this technology can be expected to produce in the years immediately ahead, and predicts what effects these changes will have on the standardization of existing languages and new languages.

4.1 Current and Future Trends in HOL Technology

The many interrelated areas of modern HOL research may be loosely classified into studies of extensibility, abstraction techniques, and ways to increase program reliability.

"Extensibility" of a programming language refers to the facilities in the language that permit the user to modify the language to suit the needs of a particular application. It offers one approach toward simultaneously achieving power, simplicity, and commonality in HOL design. In essence, the idea behind an extensible language is that a comparatively small "base" or "kernel" is provided, which includes powerful facilities for defining new data types, new operators, new syntax, and perhaps new structures for flow of control. Using these facilities, the programmer can extend the base language by defining constructs which are tailored to the intended application area and which enable him to express his solution clearly in algorithms that are not cluttered with low-level details. Since any programmer's extension is derived from the common base, the same compiler may be adapted for any extension. So, instead of having to use an entirely different language for a different application area, or having to invent a totally new language when a new application area is discovered, or having to use a large, complicated "all-purpose" language, the programmer defines whatever language tools he needs, using a common base.

More than two dozen extensible languages have been proposed and/or implemented so far [Standish 1975]. These activities reached their peak intensity in the years around 1969-1970, when several major conferences were devoted exclusively to extensibility. Development of extensible languages continues today, but the HOL revolution that some persons had predicted never took place. One reason for this is that in order to provide extension facilities powerful enough to derive languages suitable for a wide variety of specialized applications, the base language would have to be made excessively complex. Research into methods of extensibility continues, however, and the research has yielded numerous benefits for modern design of general high-order languages of modest extension capabilities.

Another trend in HOL design has been the development of language tools to support what has come to be referred to as "structured programming." This effort originated with the design of control structures which minimize the need for GO TO statements. Various iterative and conditional constructs were designed which permit a given programming problem to be solved in a way which approximates the most natural conceptualization of it. These, together with sufficient "abstraction" facilities, permit a programming problem to be solved in a "top-down" fashion, i.e., beginning with a top-level formulation of the solution expressed in terms of a number of lower-level abstractions, and so on down to the lowest level, which is written using the primitive structures in the language. This orderly decomposition of large programming problems into smaller ones facilitates the programming of large systems of programs. Each "module" in the larger system has a clearly-defined purpose within the system, and has a clearly-defined interface. It can be written by a programmer who is given instructions sufficient to the scope of his particular task.

Much of recent HOL technology has been devoted to developing the best language tools to support the abstractions necessary for structuring

of programs in this way. The "procedural" abstractions were the first to be designed in HOLs, and they date back to FORTRAN. However, a more recent approach to abstractions has been the idea of abstract data types. New languages such as SIMULA [Dahl et. al. 1970], CLU [Liskov 1976], ALPHARD [Wulf 1974a], and CS-4 [Brosgol et. al. 1975] provide facilities for the creation of "data abstractions" that are characterized completely by a set of operations that may be performed on objects of that type. The representation of the type and the definitions for the associated operations are grouped together and are partitioned from the rest of the program, usually in such a way that the user of the new type has access only to the operations which characterize that type.

Abstract data types, when used with other techniques of abstraction, add to the programmer's ability to divide the problem at hand into neatly factorable segments. They are therefore ideally suited to the construction of large software systems of the size and complexity that have been projected for the next decade [Kosy 1974].

Another growing concern among HOL designers is how to create languages which maximize software reliability -- i.e., the extent to which the software can be expected to perform its intended functions satisfactorily [Reifer 1975]. Abstract data types aid reliability by reducing the amount of information that the writer of one portion of a system needs to know about how the other portions of the system are written. Other characteristics of new languages designed for high reliability include (1) a small set of consistent rules for each language feature, (2) compiler enforcement of these rules, (3) a small number of features, so that the programmer can easily master the entire language, and (4) security, even at the price of language flexibility. For example, close compiler-enforced type-checking of all assignments to data objects and prohibition of equivalence types and untyped pointers are two ways of increasing security at a modest sacrifice of power and flexibility.

The projected increase in the size, complexity, and cost of soft-

ware systems indicate that now and in the future, high-order languages must be designed with explicit concern for reliability [Gannon and Horning 1975].

The current trend toward the design of smaller HOLs is perpetuated not only by a concern for reliability, but also by an increased understanding of the psychology of the programming process. A small language can be mastered more easily than a large language that has lots of complex and interrelated features. A small language frees the programmer to focus his attention on the problem to be solved, rather than on his tool for solving the problem. Exactly what language features are the best ones from a psychological point of view is the subject of a considerable amount of current research. Results are becoming available, but it will be a number of years before it is known just what is the optimum selection of even the language features now implemented in current languages.

The various trends in current HOL technology are exemplified in CS-4, a language developed for the U. S. Navy. CS-4 is an extensible language originally documented in 1973 [Miller et. al. 1973] and thoroughly revised in 1975 [Brosgol et. al. 1975] to admit data abstractions and more powerful extension facilities. However, the kernel facilities contain a large number of powerful features which have yet to be tested and which entail a number of theoretical design issues which have yet to be resolved. As a result, a subset of CS-4 has been proposed with only modest extension facilities and a much smaller number of features and rules. The subset language is in keeping with the current trend toward small, reliable languages with modest yet modern abstraction facilities. The full CS-4 language is an ongoing research project, continuing the study of extensibility, abstraction techniques, and how to achieve these ends in a manner consistent with the requirement that the language be simple and compact.

Advancements in HOL technology come slowly, because (among other reasons) the interrelationships between the features of a language are

intricate and subtle. Adding, modifying, or deleting any feature has ramifications which affect other features of the language, often in ways that are not immediately obvious. This point was emphasized at the Fall Joint Computer Conference, 1968:

> Constructs interact with each other, and the addition of a single feature may, in a bad case, double the size of the translator. The potentially exponential growth of translator size with increasing language complexity has two important implications. First, a translation method which works well for deliberately simple test languages or machines may fail for practical languages or existing machines. Second, an elaborate language (we have in mind the full PL/I) may need a much larger translator than needed by several smaller languages which have in aggregate the same features. [McKeeman et. al. 1968]

Most languages strike some sort of compromise, a tradeoff between simplicity and power, between security and flexibility. One cannot readily write down a complete list of "desirable" language characteristics and proceed to find or design a language that meets all of those criteria.

A list of characteristics has been proposed by the Department of Defense [Tinman 1976], known as the "Tinman" requirements for a common high-order programming language. The characteristics enumerated in this "Tinman" paper are worthy goals of language design, and almost every one of them, considered separately, is currently achievable. The task for a language designer is to deal with the interrelationships between the language features necessary to realize all of these characteristics, such that the simultaneous achievement of the whole set of characteristics does not entail intolerable complexity. The extent to which a language meeting the "Tinman" requirements is currently within the state of the art, and what compromises would or should be made to approximate that language, was the subject of a DoD conference of HOL specialists in September, 1976.

One prediction is fairly safe: the improvements in HOL technology that we can expect in the years ahead will be rather modest ones. There will be no "revolutions" in HOL technology in the decade ahead -- nothing comparable to the revolutions in hardware technology that have distinguished the various generations" of computers [Ralston 1973]. The breakthroughs will be minor ones, such as determining the psychologically optimum set of control structure primitives for a general-purpose HOL, or finding the most effective means for providing parallelism into a language (see the Operating Systems section of this report), or devising a means of implementing data abstractions in a way consistent with real-time requirements [Clark et. al. 1975].

Even after these improvements in HOL technology are developed, it will be several years before they will have much impact on actual programming in non-experimental situations. (We will return to this point in Section 4.3, below.) Programs for actual applications in large systems will be written in standardized languages, and it would be foolish to attempt to incorporate each minor improvement into the standards as it is developed. That would do more harm than good and would contradict the very reason for standardizing languages in the first place.

We can also see this "time lag" between HOL improvements and their migration into general use by observing the difference between the attitude toward HOL technology inside the university environment and that outside of the university. Most of the recent languages developed for the purpose of advancing the art of programming have been developed under the auspices of universities. PASCAL, CLU, ALPHARD, and SIMULA 67 are some of the noteworthy examples. The desire here is to do things the "best" way, and the risk of experimentation is slight. It is in industry, business, and government, however, that most of the world's operational software is written. Here, the motivation is somewhat different, because the risks of experimentation are higher. Programmers are loyal to the

languages with which they are familiar. Vast bodies of working software exist, and forced obsolescence is expensive. Experiments that end in failure cost not only dollars, but also company reputation and perhaps human lives. As a result, most of the world's computer programs are still written in FORTRAN or COBOL -- languages which do not even incorporate HOL improvements made in the last ten or twenty years. But the resistance to a change in languages is so great that FORTRAN and COBOL will continue to dominate HOL programming in the decade (or perhaps decades) to come.

Nevertheless, the desire to change existing languages to incorporate HOL improvements is also present. The impact that these changes will have on existing languages in the face of standardization efforts will be discussed in the following paragraphs.

4.2 Existing Languages, New Technology, and Standardization

FORTRAN

Of the most widely-used HOLs, FORTRAN is the one in which major changes will occur soonest, and, because it is a language for which a standard already exists, it illustrates some effects that changing any standard language entails. We will discuss FORTRAN at considerable length, because it illustrates many of the problems that arise when any language is standardized (or re-standardized) after being surpassed by other achievements in HOL technology.

In the United States, a national standard for FORTRAN has existed since 1966. Recently, a draft of a new FORTRAN standard has been proposed for discussion prior to adoption. In a few months, the draft proposed by the American National Standard X3J3 committee [BSR X3.9 1976] will replace the USAS X3.9-1966 standard. The stated "primary purpose" of the new standard is to promote the portability of FORTRAN programs. For this reason, all of the committee members sought to preserve the 1966 standard where possible, and to design the new standard

in such a way that it contradicts past efforts only when the improvement is of overriding importance.

One category of changes to be reflected in the new standard is merely the identification as illegal certain meaningless or near-meaningless constructs that the 1966 standard does not prohibit. (For example, out-of-bounds subscripts, negative values for I/O unit indentifiers, and redundant type-specifications are legal under the 1966 standard but will be illegal under the new standard.) These changes improve the portability of standard-conforming programs by making the criteria of the standard more strict. As will be pointed out below, making language specifications more rigorous is a trend that other languages will also follow.

The proposed FORTRAN standard also contains some new features which will increase the power and flexibility of the language and bring it more nearly up-to-date with some HOL improvements made during the last fifteen or twenty years. A character data type has been added, along with concatenation and substring operations. Arrays may have as many as seven dimensions (not just three), and the user may specify explicitly what the lower bounds are. The X3J3 committee is still in the process of revising the proposed standard. An IF-THEN-ELSE construction has recently been voted into the draft, and non-integer subscript expressions have been removed. This kind of revision process will continue for several more months, at least.

Most of these changes are <u>additions</u> to the language; programs written under the old standard will be unaffected by the additions and will be compatible with future programs written under the new standard. This goal of compatibility, although a laudable one, does entail some disadvantages. The language grows in size rather than shrinks. The new features often have generality and power that makes their usefulness overlap with that of existing features. The proposed ANSI FORTRAN illustrates this tendency. For example, the character data type will replace the Hollerith type and will offer the advantages mentioned in the preceding paragraph. However, Hollerith field descriptors will still be permitted in format statements. So, under the new standard, there will be two widely

differing ways of specifying formats: by means of character expressions and by means of format statements. There will be two ways of getting characters to appear in output: by means of character data, and by means of Hollerith field descriptors in format statements. One solution to the problem would be to remove format statements and Hollerith field descriptions from the language entirely and decree that standard programs shall handle formatting and character output by means of character expressions exclusively. That solution, however, would make obsolete almost every FORTRAN program in existence.

This tradeoff between generating incompatibilities and increasing the size of the language is one that will arise every time a standard (for any language) is revised to conform to more modern programming methods. Future standards committees, like this committee for FORTRAN, will decide on some sort of compromise.

The new ANSI FORTRAN standard will be a "lenient" standard in that it will not prescribe how a processor must respond to a program that violates the standard. A standard-conforming processor will not necessarily reject programs which the standard explicitly specifies are illegal. The existing 1966 FORTRAN standard took this approach in order not to inhibit language growth by prohibiting superset implementations. (Four years earlier, FORTRAN IV had been an outgrowth of the then-six-years-old FORTRAN II. The standards committee did not want to inhibit that kind of growth in the future.)

There are two consequences of this kind of standardization. First, programmers acquainted with one particular processor may unintentionally write non-standard programs and never realize it until either they or their programs are moved to a processor which happens to detect the error. For example, many FORTRAN programmers became accustomed to using mixed-mode arithmetic, which is illegal according to the 1966 FORTRAN standard but accepted by many processors. (Mixed-mode arithmetic will be legal

under the new standard.) So, although the proposed standard makes explicitly illegal certain possible constructions about which the 1966 standard is silent, one must keep in mind that a modification of a 1966 processor to bring it into accordance with the new standard will not necessarily entail making the processor reject the newly-illegal constructs.

The second consequence of this "lenient" standardization is that it omits from the province of standardization the highly useful area of diagnostics. In 1966, processor standardization was rejected as "premature," because "so many considerations affecting internal processor construction are involved" [USAS X3.9-1966]. Thus a precedent was set, which all ANSI language standards have followed and in all likelihood will continue to follow in the years ahead.

A more recent view is that of C. A. R. Hoare. "It should not be left to an implementation to determine error messages: that is the duty of the language standard itself" [Hoare 1973]. In the coming decade, language standards will begin to follow Hoare's recommendation. Several factors will necessitate this change: (1) Debugging involves much of the time and expense in developing software. (2) The advantages of portability are only partially achieved when a program produces a different set of diagnostics (and possibly even different behavior) for each processor. (3) The number of makes and models of computers in use has increased substantially in recent years, increasing the necessity for easier portability. (4) Diagnostic messages are frequently difficult to understand.

The 1966 FORTRAN standard allows superset implementations in order to permit further language developments. This reason, however, is even less compelling now than it was in the early years of FORTRAN. While it is certainly true that development and testing of new HOL concepts should continue, the state of the art in programming language design has progressed so far beyond FORTRAN that it is not realistic to expect any

significant developments to take place within the context provided by standard FORTRAN processors. Even for more modern languages in which it would be possible to test new HOL concepts, a standard-confirming processor for operational software is not the proper vehicle for such research. Future HOL standards should (and many non-ANSI standards will) prohibit such superset implementations.

Another trend toward the "modernization" of FORTRAN programming entails not changing FORTRAN itself, but adding to the FORTRAN compiler a preprocessor which accepts as input a "structured" form of FORTRAN and produces standard FORTRAN as output. The increasing popularity of FORTRAN preprocessors indicates both the popularity of FORTRAN and the desire on the part of users for modernization along the lines of "structured programming." Nobody knows just how many FORTRAN preprocessors are now in use. A list of over 50 currently-available FORTRAN preprocessors was published earlier this year [Reifer 1976].

There is no standard to control the development of FORTRAN preprocessors. The currently available preprocessors implement incompatible syntax and constructs, and the situation is not likely to improve. Standardized versions of structured FORTRAN constructs will come only when they are incorporated into the FORTRAN standard itself. The proposed revision of the 1966 FORTRAN is a small beginning of that process. The new FORTRAN standard will reduce somewhat the desire to modernize FORTRAN via preprocessors, and another FORTRAN committee will almost certainly be organized to present us with yet another version of FORTRAN within the next decade. Future revisions to FORTRAN will be mainly syntactic, in order to keep down the size and maintain the speed of the language and its processors. One advantage that FORTRAN has over some more modern languages like PL/I and ALGOL 68 is its small size, and a future standards committee will be reluctant to make alterations that would substantially affect that advantage.

PL/I

PL/I is a large language that was developed by IBM and an IBM user's association in the early 1960's. IBM had hoped that by combining features of FORTRAN, COBOL, ALGOL, and several other languages, a multi-application language would be created which would eventually replace both FORTRAN and COBOL. Because of IBM's power and influence, PL/I is now a well-established language, and it will continue in use throughout the next decade. It will never, however, replace FORTRAN and COBOL -- a fact that IBM finally conceded [SIGPLAN 1974]. PL/I's popularity has been limited in that compilers for it are available for only a few non-IBM machines.

An international standard for PL/I has been in preparation for several years by a joint committee (X3J1) of the European Computer Manufacturers' Association and the American National Standards Institute (ECMA/ANSI). The draft standard [BSR X3.53 1975] has now been given final approval by ANSI. The defining document uses a method of presentation more formal than that of the proposed FORTRAN standard. This fact, plus PL/I's size, resulted in a defining document that is several times the length of the draft proposed FORTRAN. The primary aim of the new PL/I standard is not to revise the language along the lines of recent HOL technology, but to formulate a precise common definition of a version of the language that is close to what is currently in use. (The new ANSI PL/I is quite close to the existing MULTICS PL/I, except that multi-tasking is absent.)

Now that the ANSI PL/I has been approved, PL/I's popularity will rise somewhat as various manufacturers develop compilers in accordance with the new standard. This development will not happen quickly, because the size of PL/I and the complexity of its many interdependent rules make writing a compiler that can be guaranteed standard-conforming a lengthy process.

However, the popularity of PL/I for the creation of large systems will eventually decline due to the large number of constructs and provisions in the language which detract from reliability. (See Section 5, below.) In short, PL/I is the product of language design during an

era before reliability became a major concern. Furthermore, the new PL/I standard places no requirements on how a processor treats illegal programs. This "permissive" standardization will detract from the reliability and portability of PL/I software even more than it does from FORTRAN's, because of the subtlety of the interactions among PL/I's many features.

Realizing that PL/I's size is one of the language's main drawbacks, the PL/I X3J1 standardization committee is now at work designing subsets of PL/I. Currently, three levels of subset are envisioned: one for general-purpose use with minicomputers, a smaller one for use on microprocessors, and a third for process control in a real-time communications environment [Frampton 1976]. The success of the PL/I subsets will depend on the way in which they are determined. PL/I is a difficult language to subset neatly, because the features of the language are quite interrelated. Also, the intricate rules governing defaults and special-case behavior are not the sort which can be unified by subsetting. As Hoare has observed of PL/I, "...The removal of unwanted features tends to remove a lot of wanted ones as well, or else to leave an untidy hole, pluggable only by complicated rules and exceptions" [Hoare 1973]. The committee will be forced to sacrifice either upward compatibility or uniformity and "cleanness" of design.

At best, the process of defining standard PL/I subsets will be lengthy and difficult. The X3J1 committee worked on the full PL/I standard for over six years, and it will be at best several more years before standard PL/I subsets are approved. A very early draft exists of the larger general-purpose subset, but it will take three more years (at a very minimum) before a standard is ready [Frampton 1976]. The other subsets will take even longer.

COBOL

COBOL is maintained by the Programming Language Committee of CODASYL (COnference on DAta SYstems Languages). This committee meets regularly and publishes a new edition of the CODASYL COBOL Journal of Development every one to three years. The most recent CODASYL COBOL Journal of Development [CODASYL 1976] reflects the CODASYL COBOL language as of January, 1976, and replaces the version of 1973.

Independent of the CODASYL Programming Language Committee is the American National Standards Institute's X3J4 Technical Committee on COBOL, whose purpose is to propose a version of COBOL to be the ANSI standard. The current ANSI standard COBOL [ANSI X3.23-1974] was adopted in 1974 and replaces a 1968 standard. ANSI standards remain in effect for at least five years, so ANSI COBOL is more stable than CODASYL COBOL.

The X3J4 committee bases its recommendations for changes to COBOL on the specifications published in the CODASYL COBOL Journal of Development. The specifications of the X3.23-1974 standard were drawn from the 1968 ANSI standard and the CODASYL Journals that were published prior to December, 1976 [ANSI X3.23-1974].

A reasonable prediction of what the next ANSI COBOL will contain can be made by examining the changes to COBOL reported in the two issues of the COBOL Journal of Development that have appeared since 1971 and the changes to COBOL that the CODASYL Programming Language Committee is now considering. These include a data base facility, a collating sequence and character set declarations, and a bit-manipulation facility, plus numerous minor changes (all reported in the 1976 journal). The CODASYL committee is currently considering adding a facility to define functions, an asynchronous processing facility, and most significantly, a number of language facilities to support structured programming. It is safe to predict, therefore, that in five years or so, ANSI COBOL will be facing the deliberations and tradeoffs that the ANSI FORTRAN committee is now facing.

Like other ANSI standards, ANSI COBOL does not specify a processor's treatment of non-standard programs.

SIMULA 67

SIMULA 67 is a general-purpose high-order language developed by the Norwegian Computing Center (NCC). It is relevant to the current discussion for two reasons: (1) it was designed by expert computer scientists who knew well the directions of future HOL technology, and (2) it is currently under the control of a successful standardization program which can serve as an example to other standards groups.

SIMULA 67 is based on ALGOL 60, but also includes facilities for data abstraction and modular program construction, as well as design characteristics to enhance reliability: type-checking, compile-time error checking, and controlled access to data and operations. In these respects, SIMULA 67 is close to the forefront of HOL technology and contains facilities and restrictions typical of languages currently being developed and languages yet to be designed. SIMULA 67 is available on most systems that support ALGOL 60.

The NCC set up the Simula Standards Group (SSG) and gave it authority over the standardization and maintenance of SIMULA. The SSG is composed of representatives of firms and organizations having responsibility for SIMULA 67 compilers. The SSG lacks the authority that a standards board established by the government or the military would have. Yet it has been remarkably successful in preventing spurious versions of the language from arising. Several factors contribute to the success of the SSG: (1) The SSG is composed of representatives of the various implementors. The fact that an implementor can influence standard SIMULA 67 makes him reluctant to devise his own non-standard version. (2) The number of representatives is small (currently less than a dozen), so the number of vested interests is low, and the proportionate weight for each member is high. (3) The SSG has been very reluctant to change SIMULA 67 from its original form.

Evidence of reason (3) above may be seen by observing how little the 1970 defining document [Dahl et. al. 1970] differs from the original 1968 version. One change that was voted into the recent version this year (release 3, January, 1976) is that the representation of abstract data types

(called "classes") may now be hidden from users. This change is in keeping with current philosophy of making data types less liable to misuse Other requested changes have been rejected as being mere minor conveniences (e.g., providing case and repeat statements). Changes currently being given serious consideration are of a minor sort, such as "whether to allow access inside a class to attributes of another instance of the class by remote access", or whether the hidden attribute should "change all attributes or only one of them if there are several attributes with the same identifier in the prefix chain" [Gazette 1976].

The SSG does little "pruning" of the language -- changes tend to be enhancements rather than changes which would conflict with existing software. Nevertheless, the success of the standardization of SIMULA 67 demonstrates an approach to standardization that should be followed for other languages. SIMULA leaves no language constructs "undefined" [Palme 1975], and unlike the ANSI standards, the SIMULA 67 standard requires that the processor reject illegal programs.

CORAL 66

CORAL 66 is another example of a successful standardization effort. The British Ministry of Defence adopted CORAL 66 as its standard language in 1972. It is a language of only modest capabilities. It lacks all but the most rudimentary extension facilities, and it lacks the modern abstraction and protection facilities of SIMULA 67. The British Royal Radar Establishment, rather than use the CORAL 66 standard, implemented and uses a version of the much more powerful ALGOL 68. Nevertheless, the standardization is a success. Use of the language is increasing, not only in the Ministry of Defence, but in industry. The Department of Industry has urged manufacturers to support it [Shorter 1976]. Twelve new CORAL 66 compilers appeared in 1975, many on non-British computers [Noyes 1976].

CORAL 66 will continue to increase in popularity in Britain, primarily because of the influence of the British government. This success is testimony

to the fact that official sanction of a standard can perpetuate and insure the popularity of a language that is adequate but not at the forefront of HOL technology.

PASCAL

PASCAL is a language that was designed by Niklaus Wirth [Jensen and Wirth 1975] at ETH (Federal Institute of Technology] in Zurich. PASCAL contains a small number of carefully-chosen (and well-chosen) features, with the meticulous elimination of features that make for either obscure rules and behavior or difficult implementations. The result is a language with modern control-flow constructs and a versatility that is remarkable in view of its small size.

PASCAL's main problem is that it lacks the support of a major computer manufacturer (cf. PL/I). Nevertheless, its popularity is increasing -- a fact that can be due to nothing but the merits of the language. It is still identified primarily with universities and the academic environment, but interest from industry will increase as more compilers become available.

PASCAL was developed first for the CDC 6000 series (1965-74). Then, in 1972-76, PASCAL became available on seven more machines. Currently, compilers are under construction for eight additional machines. There is currently no standard for PASCAL, although a formal specification does exist. Various implementations are free to make minor modifications to suit their own applications.

Because the number of features in PASCAL have been kept to a minimum, users are often tempted to correct the alleged defects by "enhancing" the language in various ways (e.g., by adding dynamic arrays). The most successful enhancement of PASCAL has been Concurrent PASCAL [Brinch Hansen 1975a], discussed below in Section 11.2.4.

PASCAL is a language that is ready for standardization but lacks the vehicle which would give authority and enforcement to a standard. Enough programming has been done in PASCAL to thoroughly test its selection of features, so standardization would not be premature. In the absence of an

enforced standard, competing dialects of PASCAL will arise, each with its own group of followers.

JOVIAL

If PASCAL is a language that is ripe for standardization, JOVIAL is a language for which standardization is overdue. JOVIAL has already proliferated itself to the point where a great deal of operational software exists in conflicting dialects. Some JOVIAL dialects, however, are more prominent and have achieved wider use than others. Of the JOVIALS in current use, J3 has the widest use and is officially defined in [AFM 100-24] (soon to become MIL STD 1588). Various dialects of J3 abound (e.g., J4, used for the Satellite Control Facility, and J3B, used for the B-1 program). JOVIAL underwent a thorough revision in 1973, and J73 was the result. The consensus was that J73 was too large, so a "level one" subset, J73/I, was created. It is ironic, and typical of the dilemmas facing standards committees, that of all the prominent JOVIAL dialects, the one that is the most up-to-date (J73/I) is the very one for which the least operational software has been written. (However, several programs -- both completed and in progress -- do use J73/I, and the list is growing. The J73/I compilers, the DAIS mission software, and the JCVS compiler verification system are noteworthy examples.)

Even within one JOVIAL dialect, minor differences exist. The version of J3 that the JOCIT compiler-implementation tool produces is slightly different from the version that is documented in AFM 100-24. The version of J73/I at Rome Air Development Center (whose defining document is soon to become MIL STD 1589) is slightly different from the DAIS J73/I.

In terms of modern priniciples of HOL design for reliability, the major defect of the various JOVIALS -- even the more modern J73/I -- is that they do not enforce a strict form of type-checking.

The future of the various dialects of JOVIAL, like that of other languages used within the Department of Defense, must await decisions from within the government regarding standardization. It is clear, though, that hard decisions will be necessary to correct the fact that standardization is overdue, and not all users will be pleased.

4.3 Future Languages and Standardization.

It is unlikely that a yet-to-be-designed general-purpose HOL will be created, implemented, and standardized in time to affect software systems in the next ten years. This is due partly to the fact that HOL technology advances slowly and no major breakthroughs are in sight. It is also due partly to the reluctance of programmers to turn their backs on languages with which they are familiar and the reluctance of program managers and project managers to make obsolete many man-years-worth of software. But it is also due to the length of time necessary to design a new language, implement it, refine the design, test the design in actual production environments, and finally standardize on a version of that language.

It may take two or three years to draw up the design of a language. Depending on the size of the language, another two years or so will probably be required to write a compiler [Bostrum 1971]. There is evidence from implementors [Brosgol 1974] that a compiler reveals design changes that are necessary in the language, so for a period of time, both the compiler and the language will be in a state of design. Even after the compiler is operational, experience in writing actual applications programs is necessary to reveal more places in the language which require change. Then the actual process of standardization itself requires as much as several years.

Experience drawn from the creation and standardization of existing languages bears out this lengthy timetable. The first draft of a FORTRAN was in 1954; the first ANSI FORTRAN standard was in 1966. PL/I was created in the early sixties; the ANSI PL/I standardization committee has been meeting since 1969 and the standard was not adopted until August, 1976. The initial version of CORAL was defined in 1962, CORAL 66 was developed in the mid-60's, and a national standard was not adopted until 1973. CORAL's "six year" standardization period has been observed with other languages as well [Enslow 1975].

Can the timetable be speeded up? Not very easily, and certainly not safely. IBM attempted to finalize a version of PL/I before implementation, and IFIP tried to do the same with ALGOL 68. Both attempts failed, and new drafts were necessary [Hoare 1973]. The danger is greater if the new language is one whose purpose is to put forward some hitherto untested and unused innovation in HOL technology.

The Department of Defense, in a recent paper [Fisher 1976] that explains the background of the DoD common programming effort and contains the "Tinman" requirements, states that it is "most desirable" that the selected common language be either an existing language or a modification of an existing language. Meeting this goal would help reduce the "innovative" nature of the selected language and therefore reduce the time necessary to obtain a secure and reliable language. However, attaining all of the "Tinman" characteristics in a single language would itself constitute an innovation.

But minor HOL innovations will continue over the next decade, and new languages will continue to appear. These will mainly be within the experimental environment of university computer science departments, and they will have little impact on military or industrial applications during that time period.

There will be one big exception to this forecast, and that will be in the area of special-purpose languages. These languages, unlike the general-purpose HOLs which are the main subject of this report, have a restricted, clearly-defined scope, and they come into existence to fill a particular specific need. As hardware technology makes available new tools (e.g., high-speed random-access memories, CRT graphics displays), new languages will arise to fit the need. The main language advances in the next decade will be in this area, not in the area of general-purpose languages (which is by now fairly stable). There will be little to inhibit proliferation of special-purpose languages, because the conservative factors of old programmers' habits and backlogs of old software do not exist yet.

4.4 Summary of HOL Technology Forecast

1) FORTRAN and COBOL will continue in wide use. This will be the case not only because of the inertia of programmers' habits and the increasingly vast amounts of existing software, but because neither language is of unreasonable size, both are fairly well-standardized and fairly adequate for their respective areas of application. Of all the computer programs in the world, though, the percentage that are written in FORTRAN and COBOL will decline somewhat. This will be true because of gains made by other languages.

2) General-purpose languages like PASCAL and SIMULA 67 with structured control-flow facilities will achieve wider use in industry as well as in academic circles. New languages will strongly resemble these languages. The changes will be in two main directions: (a) adding features for increased reliability -- i.e., stronger type-checking, more rigorous specification of what has previously been implementation-dependent (e.g., range and precision), more features which permit structuring of programs into modules and abstractions, and (b) adding features which permit adapting the language to new applications -- e.g., adding "concurrency" to PASCAL. Whether or not this prediction applies to the government depends on whether the appropriate high-level decisions are made within the branches of the government that develop and use software.

3) Interest in (and use of) large languages will decline. PL/I will, however, continue to be popular, due to inertia, support from major manufacturers (e.g., IBM and Honeywell), and the new ANSI standard.

4) Research in HOL topics (including extensibility and abstract data) will continue, but gains will be modest.

5) There will be an increase in the number of small, special-purpose languages as the need develops and as hardware develops which gives rise to specialized application areas.

All of the above changes will take place slowly, for several reasons: (1) No major breakthroughs in HOL technology are in sight, and it takes years for even substantial innovations to have a significant impact on non-experimental applications programs. (2) There is a great reluctance on

the part of programmers in industry and government to change HOLs. Past accomplishments in software development, standardization, and simple acquisition of programming skills were hard won and are not to be abandoned quickly.

5.0 PL/I

5.1 Introduction

PL/I was released by IBM in 1966 after several years of design and development effort [Sammett 1969]. Following some initial user resistance, it has become the third most commonly used HOL in industry (behind COBOL and FORTRAN) [Marcotty 1975]. The designers of PL/I studied many other languages before making the final choices of PL/I features. The block structure and attendant scope rules were derived from ALGOL; the use of picture formats to represent commercial data and the use of structures as a method of organizing data were derived from COBOL; much of the notation and many keywords and format specifiers were derived from FORTRAN. A draft standard for PL/I [BSR X3.53 1975], which has been in preparation for several years by a joint committee (X3J1) of the European Computer Manufacturers' Association and the American National Standards Institute (ECMA/ANSI), has been given final approval by ANSI. This standard, however, does not contain the compile-time macro facility nor the real-time tasking capabilities contained in IBM's PL/I. Two subcommittees are working to define a standard general purpose subset and a standard real-time (extended) subset of PL/I, but it is not clear when the results of this work will be available.

5.2 PL/I Language Characteristics

In some sense, PL/I can be considered the most powerful computer language ever developed; or ever likely to be developed. Jean Sammett states, "PL/I is very general with the widest scope of any language in this book" [Sammett 1969]. There are over 120 languages in the book.

PL/I is better than FORTRAN or COBOL because:

1. PL/I can do what both of those can.
2. It is usually easier to say it in PL/I.

3. PL/I has somewhat reasonable control structures (DO_WHILE and IF_THEN_ELSE).

4. PL/I doesn't have a vast number of silly restrictions.

5. If you ever thought you wanted it, it is likely to be in PL/I. [Holt 1973].

The initial goals for PL/I (originally called NPL) were the following:

1. To serve the need of an unusually large group of programmers. In particular, the committee constantly attempted to encompass among its users the scientific, commercial, real-time and systems programmers and to allow both the novice and the expert to find facilities at his own level.
2. To take a simple approach which would permit a natural description of programs so that few errors would be introduced during the transcription from the problem formulation into NPL.
3. To provide a programming language for contemporary (and perhaps future) computers, monitors and applications. As a frequent benchmark, the committee chose not the familiar "can we write NPL in NPL?" but "can we write, in NPL, a real-time operating system to support NPL programs (i.e., an NPL language machine)?" [Radin and Rogoway 1965].

With some reservations, we would say that these goals have been met. Of the languages evaluated by [Rubey 1968] PL/I was found most natural for all applications except a business application for which COBOL was preferred and a small scientific problem for which FORTRAN was preferred. (The languages evaluated were: PL/I, JOVIAL, FORTRAN, and COBOL.) PL/I was used as the implementation language for MULTICS, a large time-shared computer utility [Corbato 1969]. PL/I was used successfully to implement the New York Times information bank, a very large data bank/information retrieval system [Baker 1972]. PL/I is used as a teaching language for computer programming [Holt 1973 and many others].

The following sections comment on these various applications of PL/I.

5.2.1 PL/I and MULTICS [Corbato 1969 and David 1968]

The MULTICS project was an attempt to advance the state-of-the-art of computer time sharing operating systems on several fronts simultaneously. The design team decided to use a high-order language to achieve higher

productivity and lower error rates, ease the training and transfer of programmers, and allow a certain degree of machine independence. PL/I was chosen from among the available languages because of the richness of data structuring constructs, the possibility of separately compiling routines, and the degree of machine independence. What was the experience with PL/I? Good enough that it would have been used again [Corbato 1969, Graham 1968]. However, the PL/I used had no multitasking, no controlled storage, no I/O, no picture attributes and other COBOL features, and no complex variables. Simply listed here are some of the findings:

> PL/I has intrinsic implementation problems...it suffers from being designed without well-formed plans for a systematic implementation. ...
>
> The language has a lot of assumptions in it bout what kind of an environment it is going to be in. ...
>
> One of our major sources of residual trouble is that a lot of bugs have been caused by mismatched declarations, getting parameters in a calling sequence inverted, getting argument types in calls mixed up, all clerical errors in which the language gives you no help. ...
>
> We find that a typical good systems programmer produces on his first try EPL [Early PL/I]-generated object code which is perhaps 5 to 10 times as poor as hand code. ...The reason for the factor of 5 or 10 seems to be principally that programmers don't always realize the mechanisms they are triggering off when they write something down. The usual pattern when one writes a program is to think of four or five ways that one can write out a part of an algorithm and to pick one of them on the basis of knowing which way works out best [Corbato 1969].

This last point was documented with the improvements in certain modules by [David 1968]:

Module	Size Improvement	Performance Improvement	Effort
Page Fault Mechanism	26/1	50/1	3 man months
Interprocess Communication	20/1	40/1	2 man months
Segment Management	10/1	20/1	1/2 man month
Editor	16/1	25/1	1/2 man month
I/O	4/1	8/1	3 man months

The small effort required to effect great improvements speaks well for the flexibility achieved through use of PL/I. Remember that the

original programs so dramatically improved upon were also written in PL/I. That a good systems programmer must think of 4 or 5 alternative approaches to a problem before selecting one and that he must know which works out best or pay the penalty of code which is worse by factors of 5 or 10 suggests to us that PL/I is not a natural language in which to program operating systems.

5.2.2 New York Times Information Bank [Baker 1972a & b]

This system (now operational) gives New York Times personnel and other subscribers access to articles from the New York Times and other periodicals through a thesaurus and abstracts. The system was designed and implemented by the IBM Federal Systems Division and was used to evaluate the feasability of Chief Programmer Teams as a means of increasing the productivity of programmers and the quality of the produced systems. Besides the Chief Programmer Team approach, this project also used top-down structured programming and a program library system. The information bank was produced in 11 man-years with high productivity of the programmers being realized. Acceptance testing found 21 errors (approximately one error per six man-months) and subsequent operation found another 25 errors (approximately one error per five man-months) [Baker 1972b].

The effect of using PL/I may not be separated from the effects produced by using the Chief Programmer Team approach with top-down structured programming. We can note that this team restricted the use of PL/I control structures to the following:

a) Sequence;
b) IF THEN ELSE;
c) Iterative DO with or without WHILE; and
d) A simulated CASE using an array of label constants and subscripted GOTO. The source code was formatted so that indentation showed nesting depth [Baker 1972a].

Thus, a severely restricted subset of PL/I was used in a highly structured manner. The New York Times Information Bank may serve as an example of the restrictions necessary to use PL/I effectively to produce quality systems.

5.2.3 PL/I for Teaching Programming

Opinion on PL/I's suitability as a medium for teaching computer programming differs widely. "My own experience with teaching PL/I to students with a wide range of education (high school diploma through Ph.D.) has convinced me that it is easy to learn the language, provided that no attempt is made to learn the whole of PL/I in a short time. It is also an excellent language for teaching an introduction to programming" [Marcotty 1975]. "PL/I has some unfortunate characteristics when viewed as an instructional vehicle. In spite of claims to the contrary it is not adequately modular; too much must be known before one really understands certain constructions. ...We have restricted attention to a small subset and have tried to adopt conventions that will minimize the intrusions from unseen portions of the language" [Conway and Gries 1973]. The [Conway and Gries 1973] book uses a proper subset of PL/I.

After stating: "There is no doubt in this author's mind (honestly!) that PL/I, compared to FORTRAN, is a nicer, easier, more advanced and better language to teach and use.", [Holt 1973] concludes that: "We should not teach (or use) all of PL/I"....and that "We should not be lulled into using PL/I defaults. ..." [Holt 1973] offers the following rules: 1) Use no abbreviations; 2) Declare all variables and declare them at the beginning of a block; 3) Use no GOTO statements; 4) Use only a simple form of iterative DO; 5) and 6) Do not use the fixed point division operator [i.e., /] or the exponentiation operator (**); 7) and 8) Do not use the BINARY attribute or negative scaling factors; 9) Use bit strings only of length one; 10) Do not use DATA directed input/output; 11) Use no implicit conversion between strings and numbers; 12) Use no ON conditions; and 13) Adhere strictly to a set of paragraphing rules.

We do not necessarily agree with all of Professor Holt's restrictions, but we do agree that, to be used for the production of reliable software, PL/I must be restricted.

5.3 Air Force Language Requirements

The following systems requirements are summarized from [LaPadula and Loring 1976]:

Operational Flight Programs

- very efficient memory usage
- very efficient run time execution
- high reliability
- high reliability in a degraded environment
- prompt real time response

Command and Control

- efficient memory usage
- efficient run-time execution
- compiler efficiency may be important

Communications

- real-time responsiveness
- high reliability
- efficient memory utilization and run-time execution

Range Support

- reliability
- efficient run-time execution

Automatic Test Equipment

- efficient compilation

Simulator and Trainer

- efficient run-time execution

The systems requirements are then: reliability, efficient memory usage, efficient run-time execution, responsiveness in real time, and possibly efficient compilation. Because many of the Air Force systems have life-times of ten years or more, we will include ease of maintenance and modification in the requirements.

These are systems requirements, not language requirements per se; however, any HOL selected for an application must contribute significantly

to meeting these requirements. The following sections evaluate how PL/I does, or could, contribute to meeting the requirements.

5.3.1 Reliability

The language aspect most directly concerned with system reliability is security [Hoare 1974b]. PL/I has intrinsic weaknesses in the area of security. These result from the emphasis on "power" and the PL/I idea of "modularity"--that a programmer need know only the subset of the language required for his immediate application. The error checking for PL/I is not strict: "If the conceptual interpretation rejects a program-run..., then any interpretation by the implementation conforms" [BSR X3.53, Section 1.2(1) Our emphasis]. The freedom offered by this weak security requirement allows extensions to PL/I, allows optimizations of PL/I programs to omit certain run-time checks and makes possible the use of PL/I in machines which cannot support a full run-time environment for PL/I.

To make the following areas secure would require expensive run-time checking:

1) Checking the restrictions on GOTOs
   The only real restriction is that one may not jump in the middle of an iterative group. Because PL/I has label variables this restriction only be checked at run-time.
2) Checking the restrictions on pointer variables.
   These restrictions require that the object syntactically pointed to resemble (i.e., have the same type, same structure, same number of dimensions, etc.) the object dynamically pointed to. Storage for the information required to check this condition as well as the execution time to check it make checking of pointer variables very expensive.
3) Checking that all variables are initialized before their first reference. An obvious restriction for which it is generally very difficult to test.

4) Checking that argument types agree with formal parameter types for external procedures. This check can be made at compile-time for internal procedures but must be delayed until link time or run-time for external procedures.

The major difficulty with PL/I is the default conventions and the implicit conversions; the very portions of the language restricted by [Holt 1973] and [Conway and Gries 1973]. The following comments from [Tinman 1976] express the DoD's requirement for a secure language:

- Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations [Tinman 1976 E3].
- Pointer variables will be as safe in their use as are any other variables [Tinman 1976 D6].
- Implicit type conversions which represent changes in the value of data items without an explicit indicator in the program, are not only error prone but can result in run time overhead [Tinman 1976 B8].

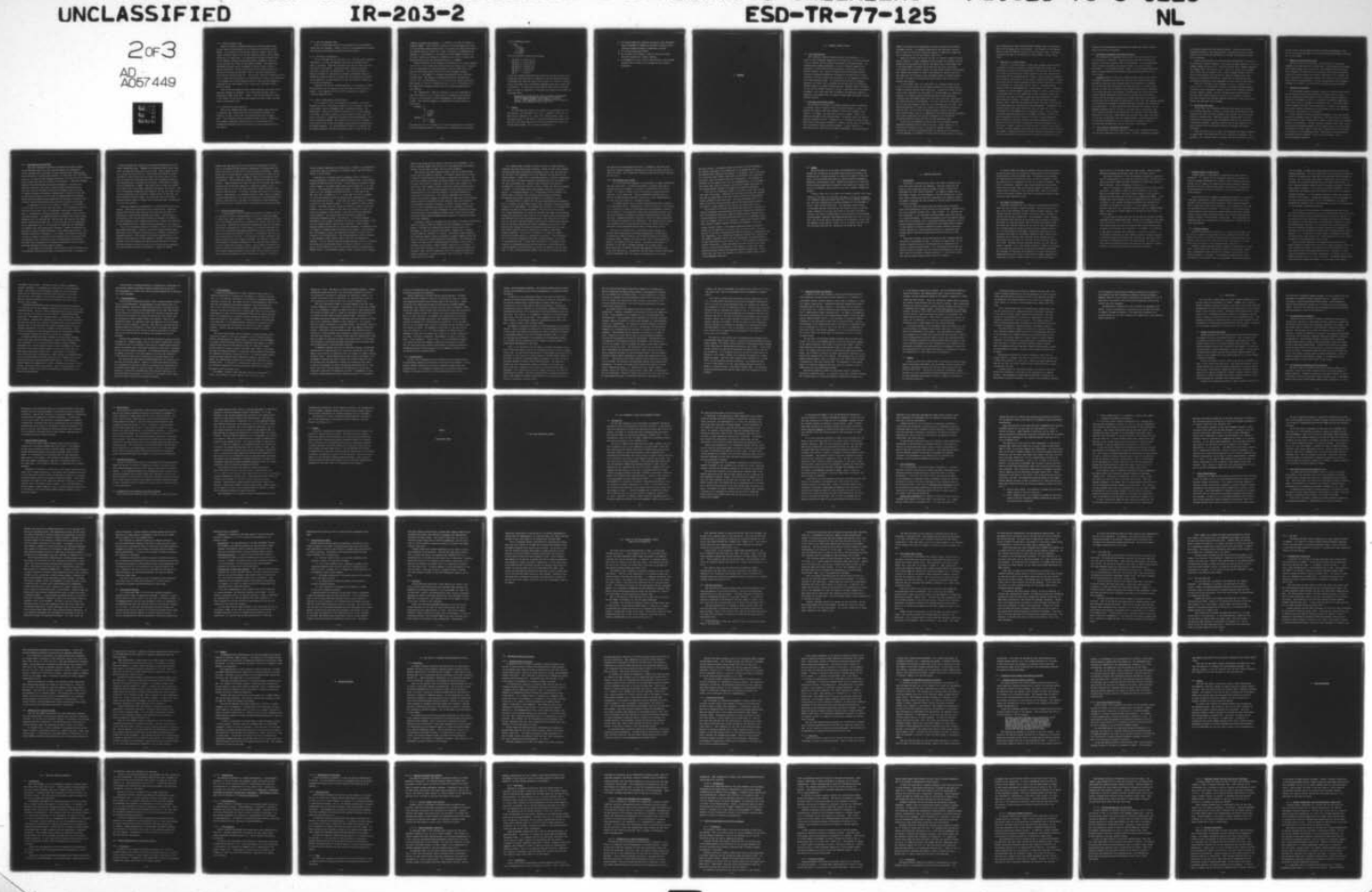
The points concerning security are best made in [Hoare 1974b] with comments directed against the PL/I approach:

> The objective of security has also been widely ignored; it is replaced by the technique of the debugging compiler, which produces object code containing many checks against programming error, achieved perhaps by partial or total interpretive execution. But this approach has several practical disadvantages. For example, the debugging compiler and the standard compiler are often not equally reliable. Even if they are, it is difficult to guarantee that they will give the same results, especially on a subtly incorrect program; and when they do not, there is nothing to help the programmer find the mistake. ...Finally, it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous [Hoare 1973b].

While it is certainly true that reliable software can be written in PL/I, the language does not give as much aid in the development of reliable software as many of the more recently developed languages such as PASCAL or SIMULA-67.

AD-A057 449 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2
HIGH-ORDER LANGUAGE TECHNOLOGY EVALUATION.(U)
OCT 76 T A DREISBACH, J L FELTY, I GREENBERG F19628-76-C-0225
UNCLASSIFIED IR-203-2 ESD-TR-77-125 NL

2 OF 3
AD
AD57449



5.3.2 Efficient Memory Usage

There is certainly no absolute measure of efficient memory usage. We shall mention some features of PL/I affecting the efficient use of memory. PL/I requires a large run-time system to support, among other things, different types of I/O--stream, sequential record, direct access record with or without keys. PL/I programs may allocate storage by any of four schemes, requiring sophisticated memory management which can only be provided by a large run-time support package. The ALIGNED and UNALIGNED attributes allow a programmer to specify certain kinds of memory optimization. The block structuring of PL/I means that only the storage in use should be allocated. Most of the above operations are clear from the program source text. The implicit conversion operations of PL/I can be invoked through ignorance or inattentiveness and may be very expensive. If an argument and a formal parameter differ in type then the argument is converted and the value passed; for an array this could waste a large amount of memory.

Array slices are passed by value, which could lead to an error if the subroutine changes its parameter. PL/I offers many features, such as assignment BY NAME, which look "cheap" in the source program but invoke a large amount of code and which would possibly be done a better way were they not encouraged by PL/I.

5.3.3 Efficient Run-Time Execution

Many of the comments made above concerning memory usage apply also to execution time, although the implicit conversion mechanisms, for instance, can have a worse effect on execution speed than on memory usage. PL/I structures may be of different forms and still "compatible" for arithmetic operations (see Section 5.3.5); the building of the intermediate structures used for the results of these operations can be very costly.

5.3.4 Real Time Responsiveness

This is a requirement mainly of the hardware and of the executive, rather than the languages. However, were PL/I the implementation language, the efficiency considerations detailed above would apply.

5.3.5 Efficiency of Compilation

PL/I's size makes for very difficult (i.e., time and space consuming) compilation. PL/I has 188 keywords, which are not reserved, and 87 built in functions. Where most languages have types or modes, PL/I has attributes--54 attributes. Identifiers may be assigned attributes explicitly, implicitly, or contextually. The rules for assigning default attributes to identifiers are complex; programmers would very seldom have reason to explicitly declare all the appropriate attributes for an identifier, so the attribute list must be completed through use of the default mechanism.

Generating the code for BY NAME assignment for operations on arrays and structures, and for operations with sub arrays or array slices is complex. PL/I compilers for the full PL/I language will always be large and relatively slow.

5.3.6 Ease of Maintenance and Modification

It is possible to write very readable PL/I programs. We believe programs written in a well-defined subset of PL/I are easier to maintain than programs written in FORTRAN, COBOL, or assembly language. The suggestions in [Holt 1973] certainly contribute to very clear programs if followed. The block structure, the free format of source code, and size of identifiers allowed are all factors allowing programmers to write easily maintained programs. The difficulty is that the PL/I language itself does not and, within the standard, cannot enforce any disciplined programming approach. If a program manager decides, for instance, that all keywords should be treated as reserved, there is no way to ask the

compiler to enforce that decision. In addition, PL/I does not support a compiled COMPOOL. Such a facility would aid in the development and control of large software systems where pieces of the system are compiled separately. The INCLUDE facility of IBM's PL/I (which is defined to be implementation-dependent in the standard) in conjunction with the EXTERNAL attribute can be used to achieve the effects of a COMPOOL but at a cost in compile-time efficiency and without the guaranteed security of a COMPOOL.

Another difficulty is the extremely wide definition of a valid statement. "DO I=0, I=1;" will iterate the do-group twice, each time with I=0, because "I=1" is interpreted as a Boolean expression which is false [Holt 1973]. Note that this might easily be written for the presumably intended "DO I=0, 1;". The reader of a program containing this error could make the same error. The reader of a program could also be confused about 'I=I=0;' (is 1 if I=0 otherwise 0) or 'IF I=I=0' (will always execute the ELSE clause) or "I=1<2<3;" (which sets I=1) or "I=3<2<1;" which sets I=1). All these seeming anomalies result from PL/I's great permissiveness in type checking.

The "compatibility" [BSR X3.53 Section 9.1.14] and "promotability" [BSR X3.53 Section 7.5.3.1] rules of PL/I are examples of the compiler going to a lot of trouble to allow programmers to get themselves into trouble. The following example will demonstrate the arcane nature of these rules:

```
DECLARE 1 A,
          2 B,
            3  C FLOAT,
            3  D FLOAT,
          2 E  FLOAT;
DECLARE 1 X,
          2 Y (3) FLOAT,
          2 Z     FLOAT;
```

The expression "A + X" is valid (i.e., the two structures are "compatible") and the following structure is generated as an immediate result generated

as an intermediate result:

```
1 tmp,
  2 BY (3),
    3 C FLOAT
    3 D FLOAT,
  2 EZ FLOAT;
```

and this structure contains the following:

```
tmp.BY(1).C = A.B.C.+X.Y(1)
tmp.BY(1).D = A.B.D.+X.Y(1)
tmp.BY(2).C = A.B,C.+X.Y(2)
tmp.BY(2).D = A.B.D+X.Y(2)
tmp.BY(3).C = A.B,C+X.Y(3)
tmp.BY(3).D = A.B.D+X.Y(3)
tmp.EZ      = A.E+X.Z
```

In the PL/I standard terminology the A-structure and the X-structure are "compatible" and either may be "promoted" to the intermediate structure. A programmer may learn what these rules produce for a particular combination of structures, but a maintenance programmer would have a great deal of difficulty. If one structure were modified then the two structures might no longer be compatible, or they might be compatible but produce an unintended result.

> One has only to consider the value of reliable, maintainable, readable programs to see that the compiler's evaluation [e.g. NO ERRORS DETECTED], while honest, is much too lenient. Many things which are not errors to a PL/I compiler are errors in programming style" [Holt 1973].

5.4 Summary

PL/I has many shortcomings as a language for programming Air Force applications. The proponents of PL/I (e.g., [Marcotty 1975]) and its more neutral evaluators (e.g., [Holt 1973] [Corbato 1969]) agree, to a surprising extent, on the nature of these shortcomings and dispute only the form of any modifications or restrictions and who should make and enforce these changes. PL/I's major shortcomings are:

1) Its complex defaulting strategy and implicit data conversions give a programmer no feedback on potential or actual errors involving misspelled words or misdeclared variables.
2) PL/I allows unrestricted GOTO's.
3) By its nature PL/I requires a compiler larger and slower than that required by a simpler language.
4) A programmer will tend to write less efficient, less reliable programs than he would with a language enforcing greater discipline.

B. Hardware

6.0 HARDWARE SUPPORT OF HOLs

6.1 Cost Considerations

Until recently, computer hardware has been very expensive. In order to offer reasonable capability for the high cost, it was the practice to try to exploit the last possible microsecond of processing speed. Any tradeoffs that reduced performance in favor of software compatibility, portability, and reliability looked economically unattractive and were usually rejected. Now, however, hardware costs have decreased whereas the cost of producing software has increased to the point where software production costs have become major bottlenecks in the completion of nearly every large software system. To counteract the high cost of software, designers are learning how to decrease the software cost by introducing hardware support for common software problems. Hardware support increases the cost of the hardware required to achieve a specified level of processing throughput, but on the other hand, introducing the additional support in hardware is much cheaper than including the same checks and support through software routines.

6.2 Efficiency of the Total System

The resultant efficiency of a hardware supported system is difficult to evaluate, since it requires viewing the entire system as a whole -- an approach not usually taken when comparing various CPU processing rates. Consider, for example, the efficiency and cost of memory. Memory is an expensive component of any computer system. Increasing either its speed or capacity increases the cost. Frequently, it is convenient to have a fixed upper bound to the amount of memory in a system. As that limit is approached, the value placed on memory savings increases. In fact, memory size constraints can be very costly to software development. As the constraints

imposed by the machine are approached, the cost of producing the software increases violently. To program an application that absolutely requires 90% of either the instruction execution capacity or memory size causes the cost of production to more than double [Boehm 1973].

Memory utilization also affects the cost of a system in terms of memory channel capacity. If only infrequent access to memory is necessary, but the access must be performed very rapidly, the memory cost is much higher than if the same number of accesses can be done over the total time period. Since memory cost is directly related to access speed, processor architectures that require the processor to spend a large amount of time per word read from memory are not favorable, since the memory channel is dormant much of the time. For means of increasing the memory channel use, see Section 7, Computer Parallelism.

On the hardware level, the "efficiency" of a computer system is the degree of utilization that the functional components of the system achieve. Larger utilization implies higher efficiency. An additional figure of merit that is often referred to as "efficiency" is the throughput of a machine as compared to a different machine constructed from the same technology. A machine using the highest speed logic and a very "efficient" architecture can achieve one level of performance. If a less "efficient" architecture is built of the same component technology, the performance will be lower. The second machine may be of lower cost, smaller size, and higher reliability than the first, but the first can always be made to process more data. This figure of merit, then, is a comparison of the maximum possible performance with a given technology level and the performance of an alternative architecture machine constructed from the same technology level.

In the past, the emphasis has been on deriving the maximal machine performance possible with the state of the art. Today, however, it is attractive to construct a fairly "inefficient" machine (implying "inefficient" use of the technology level) if that machine can reduce the cost and complexity of software. The fact that the actual gates in the system

are not operating at their maximum possible frequency does not necessarily imply that the total system is ineffectual. The direction of the hardware industry today is toward the reduction of the figure of merit. Burroughs Corporation has been on the forefront of this movement. [Wilner 1972a, 1972b].

6.3 Efficiency vs. Program Errors

The goal of providing support of high order languages through particular machine structures represents a tradeoff between efficiency (both technology utilization and processing power), and the total cost of producing a system, maintaining that system through its lifetime, and modifying that system when the requirements have changed. Certain aspects of programming introduce a large portion of the errors in functioning systems. Usually, these errors are an incorrect assumption about either an operation provided by the language, or a previously written routine. Interfaces between separately written code segments are a likely source of errors. Although it is possible for the programming language compiler to insert instructions in the generated code to verify assumptions at every transition across code interfaces, such code introduces a great many extra operations to execute while running the program. Programmers often find the extra overhead annoying, and in time-critical applications where the limits of the machine are already approached, may make the problem unworkable. Consequently, the compiler usually omits these checks and places the burden of proof of correctness on the programmer and the acceptance tester. Since programs are rarely completely tested due to the number of possible control flow paths, assumptions that are incorrect are likely to remain undetected. When the system is in actual operation, erroneous assumptions result in incorrect actions which often go undetected, causing additional errors at a later time. Clearly, finding the original cause of the error can be very difficult and time consuming, increasing the cost of producing and correcting the program. As mentioned earlier, programs approaching the machine limits cost much more which is partially due to the necessary elimination of the

interface verifications that would otherwise increase the resource demands beyond the limits of the machine.

6.4 Advantages of Hardware Interface Verification

The alternative to expensive checks in the instruction stream for correct interface is to provide a machine structure such that the checks are implicit in the basic operating mode of the hardware. It can be made impossible to either incorrectly use a piece of data, or to call a routine erroneously. If an erroneous attempt is made, it can be detected immediately before it causes confusing and possibly dangerous faulty operation of the entire system.

Software support of high order language features requires that several environment maintenance functions and operation verification functions be executed for each high order language source level operation or statement. The instructions do not necessarily contribute anything to the applications program, but they also can expend as many machine instructions to implement as the useful portion of the program. By eliminating the routine high level language induced operations from the program, perhaps 50% of the machine instructions can be saved, although this increases the potential for undiscovered error and the frequency and persistence of bugs (note that assembly language is efficient mainly because the assembly language programmer does not perform these operations). If these same maintenance functions are encoded into the hardware operations and are included in the order code of the machine implicitly, that 50% overhead is reduced by about 50%, causing a total throughput degradation of only 25%. These numbers are not meant to be exact, but rather to show that the expensive run-time overhead needed to provide maximum checking and program debuggability can be reduced by a significant amount by incorporating the operations into the order code of the machine.

6.5 Short Term vs. Long-Range Requirements

Any discussion of hardware support for high order languages must recognice that HOL technology is not fully mature. To date, programming languages

and machine architectures have evolved together. Looking at the current language structures in the research stage, it appears that there is great potential in the continued development of computer architectures to match the new languages.

Several such languages are currently under investigation in the university/industrial research community. Among them are ACTOR [Hewitt et. al. 1973], SCHEME [Sussman and Steele 1975, Steele and Sussman 1976], CONNIVER [McDermott and Sussman 1974], CS-4 [Brosgol et. al. 1975], CLU [Liskov 1976], ALPHARD [Wulf 1974a], and ECL [Wegbreit 1971]. These languages differ greatly in many of their features, but they have at least one common thread, specifically to concentrate attention on a major area of programming errors--specification, definition, and enforcement of basic primitive data behavior.

Since the new higher level languages are still in research, it is likely to be 2 to 3 years before a suitable programming language emerges. After the initial development, another 4 years will be required to develop suitable compiler systems. Finally, concurrent with the development of the high level language compilers, work will be in progress to provide additional features and alternate computer architecture to more efficiently execute the new higher level language.

6.6 Short-Range Time Scale

Current HOLs have developed concurrently with changes in the available hardware technology. Features in current HOLs are developed from the features which were originally intended to map directly onto a particular machine's instruction set. Additional machine features such as the provision of push down stacks, have increased the efficiency of processing conventional languages. Generally, HOLs are fairly well supported by the parent machine due to the parallel evolution of hardware and software technology.

During the next six to ten years, the development of computer hardware to support HOLs more effectively will be a large area of emphasis. The emphasis of the techniques used will change with the continued evolution of

the HOL itself, with the eventual result that machine technology, after the six to ten year time span, will follow the patterns described in Section 6.8 of this report.

6.7 Summary of Short-Term Time Scale

During the next six to ten years, the requirements of high order languages will remain similar to current HOLs. The availability of cheaper higher speed hardware will allow the HOL to be supported more effectivly by the hardware, and there will be an increased use of macro instructions tailored to directly execute the primitive operations of the language. The basic instructions of the machine, though, will continue to be very direct, fixed format data manipulations. The basic characteristic of the machine language will be that it is best suited to compiler-generated programs.

6.8 Long-Range Requirements

The requirements placed on computer systems in the more distant future will be directed strongly toward supporting the HOLs that will exist. Through communication with members of the university research community--especially with members of the M.I.T. Artificial Intelligence Laboratory--and in view of work in progress at Intermetrics, Inc., a few common characteristics among future HOLs have become apparent. Notably, the languages are oriented toward the abstraction of data properties into self-contained definitions. (See Section 4 of this report).

A major similarity of data abstraction languages is that they are more suited to interpretation than to compilation. The reduced cost of hardware in the future will make it practical to interpret these higher order languages directly in the hardware. The data structure (internal binary form) of the pre-processed interpreted programs will be vaguely similar to the descriptor architectures of today, except a great deal of generalization will be made in the meaning of the tagged data types. Additionally, the addressing of data objects will be extended, in line with the retnetion rules of the programming languages.

6.8.1 Requirements of Future HOLs

High order languages as currently designed do not require sophisticated hardware support for their execution because the designers of current languages were constrained to those operations that could be implemented efficiently on the machines available. Current HOLs, ever since their inception, have been proving themselves beneficial in slowing the rise of software costs in those areas that could afford the inefficiency. High order language research has identified some areas where programming is currently very expensive. Generally, these areas include multiple representations for data items that are functionally identical, procedure entry in the case of complex environment information, implementation of operations that were not originally conceived of at the time of language design, and changing binding decisions--defining which operations are constant at execution time and which requires machine operations each time the value is to be used.

To manipulate a value, be it variable or constant, it is necessary to know a small set of things about that value. It is necessary to know the data type of the object, and it is necessary to know how to access that object. Traditionally in machine structures, the hardware is only aware of the machine address. The address provides the simplest form of data access mechanism since the functions that load and store the value are the default machine load and store operations. If the value does not fit conveniently into an addressable unit, or if the object is larger than the addressable and loadable unit, then software must intervene. Software intervention is expensive in terms of system performance, as was demonstrated earlier. Since software intervention is expensive, it is avoided, and optimizing compilers are used to attempt to map the high level operations onto the hardware with an acceptable level of efficiency, with a minimum software intervention. New higher level languages are not optimally supported by the data access mechanisms provided by current hardware designs.

In new higher order languages, the distinction between different forms of conceptually identical data types is deliberately hidden. An example of

a single conceptual data type with many machine representations is the common INTEGER data type. INTEGERs can be characterized by their range and by the operations that can be done on them. Some integers are read only, others can only be written. Some integers are shared by several processes in a way so that only one process can access them at a time. Some are monitored so that debugging traces can be produced indicating each operation that is performed on the integer. Additionally, integers have some properties in common. They cannot take on values that exceed their range; they can be added, subtracted, multiplied, and divided. They can be incremented, decremented, and compared. For all data types, there are those operations that can differ between objects of that type, and those operations that are common. The operations that are different between different forms of integers can be contained within a generalized load and store function.

Read-only integers are implemented with a load function but no store function. Write-only integers have only a store function. Monitored integers have load and store functions that cause the side effect of maintaining a record of their execution as required. The inclusion of intelligence in the load and store functions is necessary for complete and total adherence to the language specification. Without total adherence to the language definition, complete machine independence is not possible.

Thus, to do the simple operation of loading an integer value can involve many complex operations. Currently, high level language compilers make sacrifices in the interests of efficiency by not absolutely following the language definition. Great amounts of effort are expended both in the execution and construction of HOL compilers to attempt to determine those contexts in which the general load and store functions must be invoked, and those instances in which the load and store functions are isomorphic the load and store functions of the machine. Indeed, for program

segments that have all run-time characteristics determinable at compile time, it is possible through extensive computation to determine when the operations need not be done, and in that way maintain efficiency. Usually though, when a program segment includes variables whose attributes are not determinable at compile time, the compiler trades off strictness in favor of efficiency. This tradeoff may introduce program errors that are not detected. These program errors often take the form of programs that operate correctly with a particular compiler on a particular machine, but if a different compiler is utilized, making different tradeoff decisions, the programs cease operation. In this way, programs can be tied to a particular compiler. Additionally, if a different machine is used, then the tradeoffs are made differently due to the fact the operations on the new machine have different characteristics, and again, the tradeoffs are made differently. Clearly this does not increase machine transportability and often precludes it.

6.8.2 Delaying Binding Decisions

Program segments that are completely analyzable at compile time can be compiled efficiently; program segments that are not completely bound at compile time cause the compiler either to introduce assumptions that are not in the language definition or to generate very general, and thus, inefficient code. Undefined and uncontrolled assumptions cause loss of compiler and machine transportability. Unfortunately, it has been found that the best and most productive programming methodologies cause the problem to be broken into many pieces--modules, functions, or subroutines--that are completely bound only at run time. It is desirable to make a subroutine that operates correctly for all parameter bindings. For instance, integer parameters should be acceptable regardless of their declared range, packing, or other variant upon the idea of integer. Here is where the general load and store facility as discussed earlier is necessary. No currently available standard commercial machine, and certainly not any machine in use by DoD for real-time command and control applications, makes it efficient to

provide a general mechanism on procedure calls. Usually, all parameters of a given data type are converted to a standard variant, and that variant is passed to the procedure.

Converting to a standard variant is possible, but it can lead to program misbehavior in machine and compiler dependent ways. As an example, consider the IBM 360 and the CDC 6600. In both cases it is desired to pass a small integer -- an integer whose range is zero through 255 -- to a subroutine. That subroutine is required to expect an integer of any possible range, up to the implementation specified limit. In the case of the 360, that limit is probably 2 ** 31. On the 6600, the limit is probably 2 ** 48 to avoid conflict with the floating point implementation of multiplication and division. In each case, the short integer is lengthened to the size of the standard integer. It is not possible to pass a larger integer. Now assume that that integer is passed in a manner to allow the parameter value to be changed so that when the procedure is exited, the actual value passed will be changed to the value of the last assignment to the parameter. The compiler has two alternatives, passage by reference and passage by copy in and copy out. In reference passage, the address of the integer is passed to the routine, and the address is used with the standard machine load and store operations. Copyin-copyout parameter passage requires that in calling the procedure, the parameter value is assigned to a dummy variable allocated by the compiler. Upon procedure exit, the parameter value is retrieved from the special location, and is assigned to the variable passed. The decision whether to use by reference or copy parameter passing must be made at the time the subroutine is compiled. If it is compiled using by-reference passage, then it is necessary that either only the standard variant of integer be passed, or that the calling routine simulate a form of by-reference binding by generating and passing a copy. If by-reference passage is used, the value will be updated

whenever the called function assigns a new value to the parameter. If a copy is actually passed, then the value is only updated when the function returns, and the value is copied into the actual parameter.

Many current high order languages require that only standard forms of data types be passed by reference. For example, PASCAL does not allow a field of a packed record to be passed to a procedure by reference. The restriction reduces the freedom the system designer has in allocating space for data, since often it is desired to have a great many integers, all packed into a small amount of space. Inefficient operation might be tolerable in this case, but the PASCAL language (at least several particular implementations) do not allow the operation to be done automatically, regardless of the efficiency. As a result, programmers are forced to invent some alternative method of performing the same action of by-reference passage without the cooperation of the language. Such write-arounds of the programming language are expensive to develop for many reasons: the initial cost of developing the method, the cost of debugging the method in the obvious cases, the cost of debugging the subtle interactions between the write-around, the programming language itself, and the cost of transporting the usually-machine-dependent-write-around to a new machine should that become necessary.

Alternately, there are languages that will fake a by-reference binding even though the routine called was compiled with the assumption of reference passage. Although this eliminates the concern from the mind of the programmer, it does not eliminate the problem. Suppose that a variable is being passed by reference to a routine. The generally made assumption with reference passage is that the variable is updated immediately when the routine to which it is passed assigns to the parameter. The assumption is likely to be exploited in a multiprogramming environment, where several processes can be reading a single variable, waiting for it to change. If the routine which receives it as a parameter does not actually change the variable's value but only changes the value of a standard form that the compiler temporarily allocates and passes, then the desired interprocess interaction will not happen.

On a simpler scale, consider a routine receiving a copied reference parameter that ends with an error condition. Unless the high level language introduces additional inefficiency on procedure entry, exit, and error exit, the new value of the parameter will not be assigned into the value of the passed variable. Indeed, that is desirable behavior in certain circumstances, often the value of variables passed as parameters should not be altered in the case of erroneous exit. The best time to determine the action on procedure exit is when the routine is written, though, not when the procedure is called. If the procedure wants copy parameter passage, it should so specify. If reference passage (i.e., instantaneous in-place change of the passed value) is desired, then reference passage should be declared. The compiler should never change the meaning of a program on its own.

The above tends to indicate that the user should specify how parameters are to be passed, and the compiler should never transform a declared reference passage into a copy passage. This prevents the necessity of an implementation decision, but does not solve the problem. There will still be the need for in-place modification of variants on the basic data type, and the persons needing the ability will have to pay for it by going outside the language with the difficulties and cost discussed previously. The reason for this difficulty is the use of the basic machine load and store instructions to perform the load and store operations on the bits representing a data item. This does not work because the loading and storing of the data item is a much more general operation than the instruction provided by conventional machine structures. To arrive at a solution, it is helpful to adopt a machine architecture known as a descriptor architecture. This term refers not to the current technology "descriptor oriented machines", but rather to the generalized form, in which each item of data in the system is described by an interpretation-oriented data structure, which describes the relevant aspects of the data object and varies in complexity with the data type. In such a descriptor-based machine, the data item includes much more

than just the bits representing the value. In addition, each data item logically includes information describing how to get its value, and how to change its value. This information constitutes the load and store functions for that item.

6.8.3 Why Software Isn't Enough

It is not absolutely necessary for the hardware to support descriptors for each item of data. A software interpreter, or special code that is executed each time a parameter is required, could provide the facilities needed. In practice, though, the use of software to provide descriptor structure is very inefficient. Various programming languages have, in effect, included software descriptor facilities in their implementations, but they have been limited to a few special cases.

One common special case of software-supported descriptor structures is the array dope vector. Languages that provide the ability to slice arrays (e.g., the ability to pass to a *one-dimensioned* array parameter a specific row or column of a two dimensional array) must provide dope vectors, since the array selection introduces two different representations for the same data type. Languages providing the slicing facility are often praised for its inclusion and cursed for its inefficiency. The inefficiency is particularly annoying when the array-slicing features goes unused, since every reference to array parameters must be able to accept array slices).

The claim is often made that a sufficiently smart compiler can identify those places where the software implementation of descriptors is required for correct execution, and include the descriptor overhead only where needed. However, the claim is valid only if the compiler has access to the complete run-time context. The compiler must be able to determine (in the case of parameter passage) all of those places where descriptors are needed, and make some appropriate decision based on the total system view. Essentially, the code used to access each parameter cannot be determined until the compiler has examined all of the calls

upon that routine. The compiler cannot be certain of having knowledge of all possible references to a procedure and to its parameters until it has compiled every subroutine and program in the system. Only at load time, or at the point where binding commitments are guaranteed to be unviolated and sacred, could the compiler make that determination. For complete enforcement of the language rules, and for transportability, it is necessary that every item of data of interest, from the most complex table down to the simplest Boolean variable, be tagged with a software descriptor.

In the case of an operating system; it can never be said that all the calls upon it are known. Compilers are continually compiling new programs that were not known about at the time binding decisions on the operating system interface routines were made. The operating system, then, is a special case where binding decisions must be left most general since the load-time of a user program is after the OS has already begun to run.

The alternative of delaying the binding decisions until a variable is actually referenced, and maintaining that binding flexibility through the use of descriptors effectively, requires hardware support [Feustel 1973]. Although descriptor oriented machines have been built, by Burroughs Corporation, for example, there is not currently a commercially available machine that provides a complete generalized descriptor mechanism. Given the current research trends, a fully general descriptor-based continuation machine will be commercially available within the next decade. The major obstacle to the use of nonconventional machines currently is resistance in the user communities. The introduction of these new machines will require time for gradual acceptance, and for the slow displacement of current machines as they reach the end of their useful life. It will probably be through the gradula introduction of basic HOL support over the next six to ten years that will allow true HOL machines to be accepted. By that time, there should be several competing HOL machines available commercially. In the twenty-year time frame, HOL machines are likely to become a major form of user-programmable computers.

6.9 Summary

During the next six to ten years, the sophistication of computer systems will increase as the problems solved become more complex. The decreased cost of computing power will enable the tradeoff of some processor time and hardware efficiency for reduced software programming, debugging, and maintenance costs. Machine structures will be much more highly oriented toward ease of automatic code generation to support HOL and will be gradually less oriented to the clever assembly language programmer.

Gradually after six to ten years, the concepts currently under investigation as the basis of future HOLs will mature, and the new languages will gradually become more widely used (due to their superior properties). New, much more interpretively oriented processors will be developed to execute these new HOLs efficiently. The experience gained through the Burroughs machines indicates that the resulting costs for the more capable hardware will be more than offset by the gains to be made in software. The various tradeoffs made in the construction of the machine (multiprocessor, pipelined, small/medium/large scale) will still be relevant, and the actual form of the machine will be different in each case. The emphasis of each machine, though, will be to support the HOL in the most efficient practical way, considering the system as a whole.

7.0 COMPUTER PARALLELISM

7.1 Introduction

The concept of parallelism has been in the minds of computer scientists ever since there were two computers. There has always been the desire to have cooperating computer systems in order to enhance both performance and total system reliability. As computer hardware has become faster, cheaper, and less dominant in the total system cost, various other forms of parallelism have been incorporated into computer systems, and computer languages have been proposed to provide theoretical modeling of parallel processes.

Currently, there are several ways that parallelism is exploited, both on the architectural level invisible to the programmer [INFOTECH 1974], and on the language interface level. These include: (1) pipelined central processors, (2) one processor executing several instruction streams in a time-sliced manner, (3) several processors at one site sharing central memory and I/O resources, and (4) distributed computing networks involving several computers connected by comparatively slow communications channels. Of these methods of achieving cooperation between different operations, the most familiar is a single processor dividing its processing power between several instruction streams based on serial devotion to each stream.

High-order languages include parallelism in their language-supplied operations in principally two ways: the ability to spawn tasks, and the ability to specify portions of the program that the computer has the option of executing simultaneously. As in any situation where there are logically several processes that can operate simultaneously on the same data, there is the danger of resource conflicts -- a situation in which more than one process requires sole access to a resource.

In current commercial operating systems, there is usually provision for more than one job to be "executing" simultaneously, and the operating system provides protection between jobs. The operating system assumes the responsibility for preventing several jobs from attempting to write the same disk file or attempting to print to the same line printer. This level of protection, although it does isolate different jobs from conflicts with each other, does not provide any support for sharing of the internal state information of the individual jobs. There is no protection for the program variables that might be accessed by different tasks within the same instruction stream.

7.2 HOL Aspects of Parallelism

High-order languages have two alternatives when supporting multiple tasks that share access to program variables. The first is merely to pass along to the programmer the basic operating system features. These features typically allow the creation of multiple tasks to operate in parallel with the main program task. The programmer in such a language must determine which portions of the program may be executed in parallel, write them in a form that permits them to be made into separate tasks, and verify that resource conflict can not occur or that the program itself provides sufficient locks and keys to prevent resource conflict. As in all cases where the programmer inherits the responsibility for determining the safety of the program, especially where timing of task executions can be variable under different system loadings and configurations, bugs are likely to occur that can not easily be traced to their cause. Programs that were operating correctly can fail when the system is changed -- e.g., when another processor is added to the system and two tasks from the same job are executed on different processors. The programmer is faced with the alternative of always enforcing programmed operations that verify at each access to common resources that access does not cause a resource conflict, or of merely satisfying

himself that all the relevant cases have been checked. When programmers themselves determine the relevant cases, they are usually wrong.

The second alternative is for the programming language to assume for itself the responsibility for enforcing resource sharing. The monitor concept of Concurrent PASCAL provides a mechanism for controlling access to shared resources. This is discussed in Section 11.2.

Work is in progress on languages that provide direct parallel control structures. One example is ALGOL 68 [van Winjgaarden et. al. 1975]. ALGOL 68 provides direct high-order language support of the concept of parallelism by providing the HOL construct of PARBEGIN and PAREND. In the PARBEGIN form, each statement within the PARBEGIN block may be executed in parallel if the compiler and machine determine that execution-time efficiency can be gained by so doing. It is the compiler's duty to determine that there are no resource conflicts within the PARBEGIN-PAREND scope. This form is a useful addition to the explicit user control of parallel tasks.

A final form of parallelism that can be supported through high-order languages and an appropriately smart compiler is not evident at the source level of the HOL at all. Potentially, a compiler can determine through flow analysis those portions of the program that can be executed safely in parallel, and execute them in parallel. For example, it is possible in many cases to evaluate the parameters to a function in parallel. If the parameter expressions are very complex and their evaluation requires a significant amount of processor time, dividing the task among several processors can decrease the total real-time required. Of course, the total amount of processor time will be greater by the amount of overhead involved in creating and coordinating the several processes.

7.3 Hardware Aspects of Parallelism

Hardware support of HOL aspects of parallelism is split into two categories: (1) support for the independent execution of parallel instruction streams regardless of the resource sharing between instruction streams, and (2) support for protection from resource sharing conflicts. The state of the art today provides several alternatives for instruction stream parallelism, but does not provide a rich set of alternatives for resource sharing.

As mentioned earlier, hardware can support parallelism by: pipelining techniques, time sharing a single processor between several instruction streams, and by providing several loosely or tightly coupled processors. Each of these will be dealt with separately. It is important to note that the alternatives are not exclusive; it is possible to have several pipelined processors, each time-slicing between several instruction streams, tightly coupled through shared main memory and I/O devices and connected through a network to other computer installations. Each alternative has advantages that can be exploited by its inclusion into a system with the other alternatives.

7.3.1 Pipeline Machines

The first alternative is "pipelining" -- providing the hardware with the ability to execute simultaneously several instructions from the same instruction stream. The hardware determines which instructions will cause resource conflict with which others and has the responsibility for determining which instructions can be executed safely [Tate 1974].

Consider a general register machine, similar to the IBM 360 or the CDC 6600. In each case, there is potential conflict over registers. Suppose that a long-execution-time instruction is placed in the pipe that is to place its result in a register. The following instruction might require the value of that register as one of its operands. The machine obviously

must not attempt to execute the second instruction until the first has completed execution. If a third instruction did not require the results of either of the previous two, then the third instruction could be placed in the pipe. As can be seen, the potential complexity of the instruction dispatcher for a pipelined machine could grow nearly without bound. In fact, pipelined machines usually trade off potential instruction execution speed against complexity in the instruction dispatch mechanism. It is assumed that any possible instruction reordering is done by the assembly language programmer -- or the language processor -- before the program is executed on the hardware. Placing the responsibility on the programmer of the actual application program (or on the programmer of the compiler) postpones the complexity, but it does decrease the cost of the hardware and is thus probably an appropriate decision.

In the future, the architecture of pipelined machines will follow the lead of vector processors such as the CDC STAR and the CRAY-1 [Cray 1975] which achieve most of their pipelined operation through the inclusion of intrinsically parallel operations in the basic order code of the machine. The complexity of these operations can be expected to increase to include more comprehensive operations. It is unlikely that pipelined machines that do not provide intrinsically parallel operations can approach the throughput of vector processors, primarily due to effective memory channel capacity. If it is necessary to fetch, decode, and determine the safety of executing many instructions to do a vector operation, the time actually required to do the operation will be very high.

During vectored operations, the hardware easily predicts which memory operands are required and initiates the memory operations before the pipe requires the operands, thus effectively eliminating memory cycle times from the basic instruction execution time. The CRAY-1, by using pipelined implementation of vector instructions, can achieve a per vector element operation time of 12.5 ns. [Cray 1975]. As hardware becomes faster, these times will probably be reduced. In the current state, however,

the speed of light factor -- the time it takes a signal to propagate through a conductor -- is already very significant in limiting processor speeds. For a factor of four increase in speed, the size of the processor and the interconnections between functional units in the hardware will have to be reduced since the time required for signal propagation through interconnection wires is already on the order of the intrinsic logic speeds.

The problem of determining which operations can be done in parallel is a difficult one for the hardware to solve, and it adds to the cost of the resulting machine. The use of vector operations helps alleviate the problem, since the pipe can be kept full as the result of a single basic machine order code. The problem still remains, though, if pipelined order codes are attempted. Although many operations can be done with vector processing instructions that were not originally contemplated when such machines were designed, most of the operations that typical programs execute can not be reformed conveniently, if at all, in terms of vector operations. Formulation of problems as vector processes that are not intrinsically matched to the problem increases the difficulty and cost of programming.

Pipelined architectures make no demands on the high-order language itself, other than that the language must include primitives that perform array processing. It is a very difficult problem to compile FORTRAN efficiently for a vector pipelined machine, since the FORTRAN "high level" constructs of DO loops and array indexing are much lower level than the basic machine operations [Schneck 1975]. Specifically, the language must provide data types corresponding to ARRAY, VECTOR, and probably MATRIX, as well as operations on those data types that can be mapped directly and efficiently onto the stream processing basic machine. Such operations include summing the components of a vector, arithmetically operating on two vectors, and moving vectors within memory. Many of the operations of the APL language [Iverson 1962] would lend themselves to implementation on a vector pipelined machine.

The provision of arrayed operations is recognized as a requirement for any military standard programming language [LaPadula and Loring 1976]. This requirement will ensure that vector pipelined machines can be exploited by the HOL programmer.

7.3.2 Multiprogramming

The next form of parallelism usually found in modern computer installations is multiprogramming. Multiprogramming involves several jobs on a single processor, sharing the computational resource on the basis of need and utility. The principle of multiprogramming is that most jobs require a much larger wall clock time than actual CPU instruction execution time. To eliminate the problem of a single job wasting CPU time, several jobs are maintained in central memory ready to be executed. When one job encounters a delay, for instance to wait for completion of an I/O request, another job can be executed. Similarly, when there is a job bound by the execution speed of the processor, if other jobs are available for running that do not require much CPU time but do require significant I/O transactions, both the CPU and the I/O devices and channels can be fully utilized.

Although multiprogramming in its simplest form does increase the total throughput of a computer system, there is an intrinsic inefficiency involved in scheduling jobs to be executed. There must be a sophisticated operating system to determine which jobs should be executed in what order to maximize the utilization of machine resources. The scheduling problem is compounded by demands on the part of the user community or by the time constraints of the application that are not compatible with maximum resource use.

One typical case where simple multiprogramming does not suffice is in the area of real-time control. Usually in a real-time monitoring system, it is necessary for some set of jobs to be executed for a fixed amount of time some critical number of times per second. The exact nature of the scheduling constraints is dependent upon the particular application, although some work has been done at the Naval Air Development Center toward defining a single time-critical scheduler interface.

7.3.3 Multiprocessing

Although multiprogramming is a useful formalism, and in fact is a practical mechanism for increasing the throughput of a computer installation, it is not directly applicable to the problem of introducing parallelism within a single job. The reason is that in multiprogramming, each of the jobs that are scheduled and executed represents a set of resources disjoint from each of the other job's resources. Since the resources are disjoint, and since the restriction is usually made that all of the resource requirements for each job can be determined before the job begins execution, there is no possibility of a resource conflict developing between two jobs during the course of their execution. In general, though, for parallelism to be meaningful in a programming system, there must be a means by which instruction streams that are executing in parallel can share resources usefully. Usually, the main resource that must be shared is program variables.

The type of parallelism in which resources are shared between several instruction streams (in this case, processes) is called multiprocessing. Usually, multiprocessing is combined within a system with multiprogramming to provide a structure whereby several independent jobs compete for system resources when they enter the execution queue, but thereafter operate without regard for resource conflict with other jobs; however, within each job there are several processes or instruction streams competing and sharing the resources allocated as a group to the job. In a multiprocessing system, operating system functions create additional processes, terminate processes, and cause process execution to be delayed until some event occurs [Kosinski 1976]. In the case of a time-critical application, OS functions are provided to specify the real-time constraints that are to govern the processes's execution [Serlin 1972].

For example, consider a real time data acquisition and processing system. One instruction stream samples data from the input

sensors and enters that data in a queue of processing requests. Another instruction stream removes data from that queue, processes it by whatever algorithm is appropriate, and queues the computation results. Yet another instruction stream is responsible for displaying the results or for incorporating the observed data into a control feedback loop.

The first instruction stream in the above example is the only time-critical component of the system. To be certain of not losing any information, it is necessary to sample the sensors as often as they produce data. By entering the sensor data in a queue for processing, the processing step can require various times to process different data. Alternately, the processing step can be temporarily delayed by some other time-critical task that is run infrequently compared to data acquisition -- for instance, output of control information to a guidance system. The important point is that the queues must be shared between at least two separate instruction streams. Potentially there are unsafe periods, when incorrect results would occur if both processes were attempting to access the information in the queue simultaneously. To prevent unsafe access to changing data, there must be a means of interlocking access to the shared data, or of allowing the processes to interrogate and modify the system state to indicate when they are performing an uninterruptable operation.

In assembly language, and often in high-order language, interprocess communication and synchronization is carried out through the use of semaphores, which indicate the operations that are allowed on a common resource [Dijkstra 1968a]. These semaphores have been totally in the domain of the programmer to maintain and test to verify that a desired operation does not conflict with the operations currently in progress by other tasks. Allowing the programmer to control interprocess synchronization on a low level, the level of routine access and transformation of shared data, introduces a great deal of opportunity for error. Programmers can inadvertently access shared data without checking the state of the semaphore,

or they can accidentally omit the semaphore maintenance operation when dangerous operations are undertaken.

Another problem with resource sharing involves deadlock. Although deadlock detection requires a large set of restrictions, there is a hardware-supported alternative which relaxes some of the restriction. Shared variables can be manipulated through the declaration of SHARED_INTEGER, SHARED_REAL, and a shared form of every other data type of interest. These shared data types provide through their load and store functions (see Section 6, Hardware Support of HOLs) an interface that allows interlocking to be done on a very local basis. For example, the load function for a shared variable would interrogate the state of the semaphore allocated for that variable to determine if a read operation is permitted. If so, the semaphore is set to indicate that a read is in progress and the information from the variable is copied. The semaphore is then reset to indicate that the read is no longer in effect. If a write operation is needed, then the semaphore is first set to indicate that a write is to be done, locking out all reads. The write is performed. The semaphore is reset to the free state. If any operation -- read or write -- is to be performed on behalf of a process for which the semaphore indicates the operation is unsafe, then the load and/or store function could execute an operating system call postponing execution of that process until the semaphore indicates that the operation can be performed safely.

7.3.4 Multiprocessors

The fact that a programming system supports either multiprogramming or multiprocessing imposes few constraints on the form of the hardware. Usually, computer systems include only one central processing element under user program control. It is possible through the use of multiport memories and sharable I/O devices to construct a system incorporating two or more

central, user-programmed processors. The operating system has the responsibility for determining which jobs and processes are to execute on which processors.

In order not to be constrained further in the choice of which processors are assigned which processes, as well as to make the construction of the operating system easier, multiprocessor systems always are constructed with a set of functionally identical processors. Although the processors are usually identical on all interfaces, hardware and software, some machines are built to be incorporated with other machines within the same class. For instance, it is possible to construct in a multiprocessor system both a CDC 6400 and CDC 6600. Other examples can be found in the IBM 360 family. DEC provides a means of interconnecting two PDP-11 UNIBUSs, creating a PDP-11 dual processor system.

The number of processors connected in a multiprocessor is not logically limited. However, because the constraints of providing memory channels and of making the total system's maintenance and control functions not consume an unacceptable amount of the total available processor time, it is rare for more than two processors to be connected in a multiprocessor system. It is likely that this limitation will be removed, primarily because of technological developments in two areas.

First, it is now possible fairly cheaply to provide many memory access ports through the extensive use of LSI technology and the sacrifice of memory access time. By time division multiplexing a single memory port among several processors in such a way as to allow accesses to separate memory modules to take place in parallel, high-speed memory technology can be used to satisfy the memory demands of several slower speed processors. The additional systems cost for providing such a memory interface is not prohibitive, since large scale integration can be used to decrease the total number of components. For a sixteen port, eight bit memory interface, less than 100 chips would be required. The major constraint that increases the component count is the number of available interconnections for connecting the processors to the memory interface system.

Any such time division memory multiplexing scheme will introduce overhead in processing each memory access request, thus reducing the throughput of processors that are memory bound. If the performance of a single processor is compared with the performance without such a multiport memory unit, the former will certainly be lower; but due to the impact of the following second development, this loss of memory performance is counteracted for many applications.

The second trend is the reduced cost of the processor in the total system cost. Currently processors are available in single unit quantities for $20. This processor is a low speed, low cost, low performance MOS Technology 6501. This processor can be operated with a minimum of support hardware, making it possible to provide each processor of a multiprocessor system very inexpensively. The trend now is toward increasing a system's performance without increasing its price. Two dominant cost-determining elements of an integrated CPU are the size of the actual IC die, and the packaging. The packaging cost is primarily a function of the number of pins, in addition to other factors such as the environmental hardness required and the mechanical strength of the package. Although the price of packaging will be reduced by further developments, the reduction in cost will be gradual, and will probably not amount to more than a factor of two over the next ten years. The fabrication of the die is the area where improvements can be expected that will reduce the cost of the die, while increasing the amount of logic contained thereon. Electron masking, although currently mostly experimental, promises much smaller device geometries. Small device size implies reduced junction capacitance, reduced carrier storage, and generally increased speed combined with reduced power comsumption. Of course, the primary factor determining speed will be the actual technology involved, i.e., saturated bipolar, emitter coupled bipolar, bipolar I^2L (current injection logic), NMOS, CMOS, or some method yet to be developed. Generally, though, the trend of the next ten years will be toward smaller devices, and many more devices per fabricated die. As

a result, the cost of processors will probably drop to about 1/2 to 1/4 of the current cost, while the performance will be increased by a similar factor.

Since small to medium scale processors are being implemented in LSI, e.g., the Data General MicroNova and the Digital Equipment Corporation's LSI-11, the cost of mini/microprocessor CPUs (not systems) will decrease with advances in the art of integrated circuits. The cost for a given performance level will decrease by a factor of 2 to 4. The primary limiting factor on the price of mini/microcomputers will probably be the very high packaging cost due to the need for printed circuit boards, power supplies, cabinets, cables, and the other overhead introduced by the need to interconnect individual IC packages. Within ten years all low-to-medium-performance computers will probably be implemented by integrated processors (microprocessors), since it will be uneconomical for minicomputers in their current form to compete with microcomputers of the same capability using LSI technology.

The ability to use low-cost LSI to incorporate many processors into a multiprocessor system should result in many more multiple-processor multiprocessor systems. One research group doing work in multimultiprocessor systems is the C.mmp group at Carnegie-Mellon University [Wulf 1974b]. In the C.mmp system, provision is made for the connection of sixteen PDP-11's into one set of memories. It is possible for one of the CPU's to pass control information to the others, allowing the cooperation necessary for a usable operating system. Less than eight computers were connected (as of summer 1975), but the operating system was operating. One interesting item to note is that almost all of the C.mmp operating system is written in BLISS-11, a machine-oriented (high-order) language (MOL). The operating system was operational very shortly (days) after the availablilty of the hardware.

7.3.5 Federated Systems and Networks

In addition to the above methods of achieving parallelism with very tight cooperation between parallel system components, it is possible to interconnect several computer systems to allow them to cooperate in a much more loosely connected way. Whereas multiprocessor systems have several processors sharing the basic resources -- main memory and I/O devices -- of the central site, loosely coupled distributed computing networks share only a communications channel [Spoonley 1974]. The amount of information that can be transmitted over this communications channel is much less than typical computer memory traffic, so the frequency of interaction between components of the distributed network must be greatly reduced. Usually, the bandwidth of the communications channel (the amount of information that can be transmitted in a unit of time) is so low that it is impossible to share tasks and/or jobs between components; it is not usually practical for the work load to be shared among the processors as it is in a multiprocessor system.

The main uses of distributed computer networks break into two categories. The first is the ARPANET approach, where the individual computers on the network primarily share access to files and data bases. The programs performing the data manipulations reside on the various computer sites. The network serves as a means of increasing communication between people, and of making it possible to access installations connected to the network from anywhere else on the network. Such networks have very general message passing formats, making it possible for any network node to address a transmission to any other network node. The generality of the network makes it an effective communications tool, but does introduce a level of inefficiency into the transmission process.

The second category of network is the special purpose, dedicated, distributed, computing system. In a system of this type, multiple processors are interconnected by a restricted bandwidth communications channel just

as in the general communications network. But the distances between processors are shorter, the communications serve a very limited range of purposes, and the information passed by the network is generally of fixed format and fixed content. The actual information passed usually represents the results of one processor that are needed by another. These distributed networks are primarily useful in special-purpose systems where there is not a requirement for frequent changes. Typical applications are in missile guidance, navigation, and control and in signal processing systems, communication between a numeric processor and several I/O processors, and communications concentrators. Notice that in the case of a central processor and I/O processors, to in fact be a distributed network the I/O processors must not share memory with the central processor.

Distributed networks do not necessarily impact the design of a HOL directly because the communications channel looks like an I/O device to each of the processors on the network. The requirements for handling the information are not much more stringent than for any other I/O device. Certain communications protocols impose critical real-time constraints, so the programming language and operating system must be capable of providing real-time response, but the requirements on the language are not significantly different from those placed on the language in other real-time applications or other applications where it is necessary to provide the transmittal of a fixed format block of information.

7.4 Summary

During the next ten to twenty years, the theoretical limits of logic speed (using known technologies) will be reached, giving a possible performance improvement of nearly 100 over the current state of the art [Turn and Sine 1973]. The ability to fabricate increasingly complex systems on a single chip through increased integration levels and the use of hybrid packaging techniques will result in computer architectures not currently practical.

Pipelined machines will continue to dominate the high end of the numeric processing applications, and the processors will continue to be highly pipelined vector processors such as the CDC STAR and the CRAY-1. The demand for such super-processors is not likely to be exceedingly large, though, so the cost of such processors will always remain very high.

Due to the decreased cost of processing power, applications that can not now be practically computerized will be automated, especially in the areas of process control and real-time monitoring. The architecture of the processors will reflect the requirements of real-time processing (see Section 13 of this report). The computing power of today's standard medium power mainframes will be available for the price of current minicomputers (ignoring possible inflationary pressure).

Since the number of computers in service will increase, and since more information gathering applications will be implemented, there will be an increased use of computer networks and federated computing systems. It will be quite common for a large factory to contain several computers, each controlling a particular process, transferring data to each other and to a central monitoring computer through a network. Also, as communication costs come down through the development of packet switched commercial networks, nationwide intra-corporate networks will be common.

Multiprocessor systems will become increasingly common with increased capacity. Processor cost will be so low that it will be practical to devote one processor to a given task. Terminal-accessed computer systems will be built for a small number of users through providing a single processor for each user.

A possible impact on high order languages, currently not very well defined as to direction, will be due to the increased use of computer networks. New programming concepts must be developed to formalize the interactions of multiple processors in a network, and programming languages

will be required to provide more convenient input/output facilities to allow greater ease when transferring information between processors. Probably, HOLs will allow a means of specifying the programs to be run in each node of the network, much as multiple processes are written using current high order languages.

Microprocessors themselves will have no effect on programming languages. Either a microprocessor's application will be small and suited to assembly language programming, or the application complexity and processor power will be large, requiring the capability of a well designed, general purpose HOL.

8.0 INPUT/OUTPUT

One of the major problem areas in current language standardization is the HOL access to the processor I/O system. FORTRAN, one of the most standard languages currently available, falls short of providing completely transportable I/O since file conventions and random access I/O are usually unique to each operating system. Determining the optimum method of representing I/O in a standard programming language is influenced by several issues, including the actual form of the I/O on the hardware itself. Many computers provide an I/O structure that is very channel-oriented, but the requirements of multiprocessor systems and the need for simple, inexpensive I/O are gradually changing the I/O structure of new computers.

8.1 Channel-Oriented Input/Output

Channel-oriented I/O is the form provided by the IBM 360/370 series of computers and their derivatives [IBM 1974]. Each I/O operation is initiated by a special start-I/O instruction, which passes the first channel control word to a logically separate channel processor. Each channel processor is physically connected to some number of I/O devices, and each device is usually connected to only one channel. In the case of a multiprocessor system, sharing devices and/or channels between processors is very complicated, since sharing does not fit into the concept of dedicated channels.

Writing the chains of control words to utilize an I/O channel optimally is very complex and error prone. And, since each channel controls several devices, an incorrect channel program can lock out many devices from service. Additionally, scheduling optimization is usually desired within the I/O devices attached to a channel, possibly requiring dynamic alteration of channel programs during channel execution based on global system knowledge not available to the normal program.

Because of the complexity involved in using a channel structure and

the difficulty of debugging channel programs, it is not practical for the programmer to manipulate channel programs directly -- especially not from within a typical applications level program. HOLs developed for channel-oriented machines provide for I/O through calls on the operating system. The OS has the responsibility for determining the actual channel programs required to carry out the desired I/O.

8.2 OS Intervention is Expensive

Invoking the OS to do each I/O transfer introduces a great deal of inefficiency, especially in cases where the basic operations provided for the device by the OS are not the operations required by the program. The most common case of inefficient I/O processing is interactive terminal handling. Each program that is using the terminal for I/O knows the exact way it wishes the terminal to be driven. Many options are available from the OS, such as echoing, character conversion, buffering, character-by-character vs. line-oriented transfers, and many others. Operating systems usually provide a means of specifying which of several of these options a particular program needs. A problem occurs when a new program is written that wishes to use the teletype in a manner not anticipated by the OS writers. The program would often best be served by direct control of the terminal, without operating system intervention. Given the channel-oriented I/O configuration, though, it is impossible for the operating system to grant direct access to the program.

8.3 I/O Through HOL Accessable Control Registers

The PDP-11 family of computers [DEC 1973] provides a model for I/O structure in which the control registers for the I/O devices appear as memory locations. Although not permitted by the PDP-11 family, it would be possible to place the individual device controller cells in the address space of the user's virtual machine [Denning 1970, Parmelee et. al. 1972].

The ability to give exclusive control of I/O devices directly to the user, removes from the OS the requirements to provide OS routines to manipulate the devices. From the HOL, the device is controlled directly by referencing variables that are mapped onto the control registers. If the control is on too low a level, then the user or system programmer can construct a standard set of routines that can be included with the user program to provide higher level control of the devices. Placing the control in the hands of the user, and allowing the individual device control registers to be accessible as variables from the user program, eliminates some of the complexity from the operating system.

8.4 Mapping Hardware Required

Providing the complex mapping needed to map the device control registers into the program address map is not trivial. Such mappings are more nearly possible now than in previous years, partly because manufacturer-supplied mapping facilities are getting closer to those needed. It is desirable to allow various sizes for each separately-mapped entity in the program. The available sizes should range from a very large segment representing a large block of program or data, to a very small page -- on the order of one word -- to map onto the I/O control registers.

Although it is possible to place just one I/O control register within a single unit of memory allocation, if the address space consumed by that page is very large and if access to many I/O registers is needed from a program, the memory address space of the program is quickly consumed. If the memory allocation unit (page) is much larger than required to hold an I/O control register, most of the address space is wasted. It is also desirable to allocate memory in small segments in order to provide detailed specification of the sharing of data and program segments between processes in multi-processing systems.

8.5 Shared Devices

The inclusion of an intelligent controller more sophisticated than an IBM 360 style channel extends the concept of individually programmed I/O control registers to handle shared devices such as disks and drums. Generally, each operating system provides a file system which manages the storage space of the device. The file system provides protection from erroneous or malicious access to files. In practice, a program opens a file and then treats the file as if it had exclusive access to a much smaller, more specialized device. An intelligent controller, in conjunction with an operating system, can allocate special memory cells serving as device control registers. The user program would request that a particular file be opened. The operating system would determine that the access was permissible, and would make an entry in the program's map containing the new control register. The intelligent controller would receive the description of the desired file, and would prepare to access that file. The program would access the control register which was allocated by the operating system, and would manipulate that register to control the file operations.

8.6 Throughput Advantages

Placing I/O device access control in the hands of the operating system burdens the system, since each I/O operation involves an OS function call. Most operating systems typically require about one millisecond to process a request. If the operating system's only responsibility was to verify that a particular program's device access request was permissible, and if the OS was not required to verify that each individual operation through the channel was in itself correct, then the total amount of processor resources devoted to user I/O handling could be reduced from a major source of system overhead to a very minimal one.

8.7 Standardization for Tomorrow Using Today's Machines

Aside from the above hardware method of handling I/O and the corollary

I/O access through the HOL, there is a serious requirement to continue to perform I/O from the processors currently constructed. It is very desirable to use the same standard HOL interface to the complete OS I/O structure across several different OSs and processors. Unfortunately, there is no simple method of specifying I/O in an OS-independent manner. Each operating system has its own particular form of file naming convention, its own file format, and its own buffering conventions. Many of the options provided by an OS are not needed for most applications, so most programs can operate successfully with a minimal subset of OS capabilities. The most promising approach to an OS independent I/O interface is to specify a small set of operations and to provide a standard way to access that set through the HOL. (For further discussion, see Section 11.3.1) A family of compatible and incompatible sets of OS features can be defined, the program declaring which set of features it needs. Any attempt to use features outside of the selected set causes a compilation error. A program that needs capabilities beyond those provided by the set will not work on any machine-OS pair not supplying those facilities. The compiler also can determine the needed function set, and make that information available to project management personnel as an aid to transportability and for program correctness verification.

Since functions are the most powerful construct provided by most current HOLs, and since the basic nature of applications level I/O is to make calls on the operating system, the form of the I/O in the HOL is best represented by function calls. These functions can be either predefined by the language, or can be provided as user-written subroutines. These routines, however supplied and defined, would be constrained to making any necessary operating system calls. These routines can be thought of as part of the operating system if desired, in which case the operating system will appear extensible.

Each implementor of a HOL has the option of determining the exact

implementation strategy for the OS interface functions. One implementation could be assembly language routines called from the HOL; another might be direct machine instructions on a machine supplying the OS in microcode.

The selection of the set of operating system functions necessary to perform most operations is the subject of much OS research. For more details, see Section 11.3.1.

8.8 Summary

Providing a transportable standard interface to the I/O system of the machine will proceed along two fronts. The first will be to provide a more direct mapping from HOL operations to the actual machine operations through the PDP-11-style device-control-register-as-memory I/O structure. The I/O structure will be improved by the ability to map memory segments in smaller granules, and by the inclusion of intelligent controllers for disks, drums, and other common devices. The second means of providing a transportable I/O structure is by specifying a minimal set of OS functions, and making a standard HOL interface to those features. There is no direct impact on a well designed, general-purpose HOL though, since the HOL OS interface would be sufficiently general to allow efficient implementation regardless of the actual form of I/O supported by the hardware.

PART II.

APPLICATION AREAS

A. Data Base Management Systems

9.0 HOL INTERFACES TO DATA BASE MANAGEMENT SYSTEMS

9.1 Introduction

The data base approach to the development of management information systems has taken on much momentum in the past few years. As the sophistication and availability of data base technology has grown from 1970 to the present, the number of installations using integrated data base management systems (DBMS) has increased tremendously [INFOTECH 1975]. It is interesting to point out that of today's commercially successful systems, many were non-existent in 1970 [INFOTECH 1975].

The work of CODASYL's Data Base Task Group (DBTG) [CODASYL 1971] has had a tremendous effect on the commercially available DBMS. Users have been finding that the network data model of CODASYL-based systems gives them more flexibility than the hierarchical based implementations. Hierarchical-oriented DBMS, while fairly rigid in terms of data structuring, still constitute a fair share of the commercial market and as such, their use should not be totally discounted. While some data base experts feel that the success of CODASYL-type systems indicates at least a 10-year existence and are apparently the way of the future [Schubert 1974], much research has been going on in the area of relational data base design. IBM has rejected the design philosophy of CODASYL and has been a source of support for relational data base innovations. Currently, IBM is working on an experimental prototype DBMS called SYSTEM R. It is emphasized that SYSTEM R is a vehicle for research in data base architecture, and is not planned as a product [Astrahan et. al. 1976]. As for other work in the relational data base area, much research effort has been coming from the academic environs. Of the few implementions of relational systems that currently exist, we find them limited in capability [Astrahan et. al. 1976]. More work is needed to develop efficient implementations of relational DBMS with complete data base management capabilities, especially

for large scale data bases of a few billion bytes.

A comparison of the approaches to DBMS organization can be made by examining the data structures themselves (hierarchies, networks, relations, and others see [Kerschberg et. al. 1976]) and identifying the limitations that these data structures impose. By data structures, we are referring to the logical view that the interface provides to its users. For instance, hierarchically organized implementations have difficulty maintaining "many-to-many" relationships among their data entities, as these systems are based on tree structures. To achieve a many-to-many data relationship, hierarchical systems must use data duplication or other special methods [Tsichritzis and Lochovsky 1976].

Rather than concentrating on the differences between the data structures of existing and proposed DBMS, we focus our attention on the different types of interfaces that DBMS provide their users. We make a distinction between the interfaces that have been proposed as their own self-contained languages and those that have been specified as extensions to existing programming languages.

Of the interfaces that are designed as extensions to existing programming languages, we are concerned with the extent to which a data base requires a particular programming language as the host for its interface.

The data base interface or data manipulation language (DML) of the CODASYL specifications has been directed towards the use of COBOL as the host programming language; however, the specifications [CODASYL 1971] do allow for other languages to interface with the data base. According to Stacey [Stacey 1974], the CODASYL DML may be independent both in function and in syntax from the details of the host language data structures, and while it is possible to define different DML for different host languages, a valuable aid to compatability results from utilizing a common DML. Stacey shows how the CODASYL COBOL-oriented DML can be used in conjunction with FORTRAN.

A contrasting philosophy is that the DML should be designed as a specific extension to a specific programming language. In this way, the data base extensions are integrated into a programming language so that the extended language can maintain a uniform syntax. IDMS, a CODASYL-based DBMS implementation available through the Cullinane Corporation, permits any host language to access the data base facilities via a CALL-type statement while also providing specific DML interfaces for COBOL and PL/I.

Of the self-contained interfaces to DBMS, we are concerned with the degree to which data base users will not need to rely on existing programming languages. While all of hierarchical, network, and relational DBMS can be interfaced by self-contained facilities, we find that the relational DBMS have concentrated on providing powerful independent languages that perform a variety of functions on the data base for use by the programmer. The different types of relational languages have been identified and classified in [Chamberlin 1976]. The use of relational systems and their interfaces permits programmers to code certain applications without using any existing programming language, but there remain applications for which the relational languages are inadequate and the full power of a programming language is needed. Again, it must be determined whether any specific relational DBMS favors a particular language to host its interface.

A criticism of the self-contained relational interfaces has been that they are not suitable for programmer use in that they do not allow efficient access of the data base. Access to relational DBMS by their interfaces is generally accomplished by specifying some attributes of a relation and indicating values for those attributes by which one can obtain other related attributes and their values without any directions on how to traverse the data base. The fact that a programmer has no means by which he can chart his own course through the data and take

advantage of the underlying implementation makes these relational interfaces inadequate for programmers that are designing applications that have strict run-time requirements.

Because the ability of an interface to be adequate can depend on things other than efficiency, the section on data independence points to relational DBMS and their interfaces as being desirable. A further point about the relational interfaces is discussed in the section on content-addressability, where we see that technological advancements can help reduce their time inefficiencies.

In the remaining sections we focus our attention on the technological advancements of data sharing and concurrent updating, distributed processors, and distributed data bases, where we identify requirements of specific features in the data base interface. Since both self-contained and host-language dependent interfaces are embedded in programming languages, any requirements we can identify for the interfaces are relevant to the programming languages that contain them.

9.2 Data Independence

For a given program, and the data that it operates on, it would be desirable to perceive the data as a logical data model independent of the underlying physical storage structure in which that data is actually represented. In this way, the logical view of the data may be mapped to the hardware by an additional layer of software so that the accessing programs can continue to function regardless of any adjustments made to the storage structure. This notion of physical data independence permits the optimization of program performance as new storage structure techniques and new hardware technology become available, without requiring any changes to the programs themselves.

Logical data independence is a measure of how well an application is insulated from changes in the data model of the data base; i.e., schema changes [Tsichritzis and Lochovsky 1976]. Of course the removal of a

logical data unit will require that accessing applications be modified, but the addition of data units should not necessarily require programming modifications.

While the benefits to be gained from data independence are certainly extremely attractive, the user pays for these benefits in terms of reduced efficiency. These problems of efficiency are the result of mapping overhead and sub-optimal processing strategies that may be adopted when the user's awareness of the underlying physical realities is obscured [INFOTECH 1975].

The exploration of the issue of data independence in existing and proposed DBMS reveals that there are a variety of ways in which these DBMS fall short of providing complete data independence. The design philosophy of the DBTG of CODASYL [CODASYL 1971] which has spawned many implementations that are widely used commercially [Taylor and Frank 1976] has been criticized [Engles 1971] for its incompleteness in handling the problems of data independence. First of all, in order to use the DML properly, the application programmer must be aware of various decisions made by the data base administrator (DBA) such as physical record placement strategies. Thus a DBA cannot change physical placement strategies without impacting certain application programs [Taylor 1974]. Another deficiency is the result of the network data structure coupled with the abilities of the DML. The underlying design philosophy of the CODASYL network approach consists of what is referred to as "owner-coupled sets". These owner-coupled sets have the following properties [Taylor and Frank 1976]:

1) Given an owner record, it is possible to process the associated member records of that set occurrence.
2) Given a member record, it is possible to process the associated owner of that set occurrence (member records may be associated with only one set occurrence).

3) Given a member record, it is possible to process other member records in the same set occurrence.

It is the very nature of such an organization and interface that restricts the programmer to thinking in terms of a specific implementation involving stored chains of pointers throughout an owner-coupled set occurrence. Even though these chains need not physically exist, the implementation must correspond to the programmer's data model, which severely limits the degree of variation possible in the storage structure [Date and Codd 1974]. Another CODASYL feature which impedes data independence is that records may be perceived as being in a specific order. The problems of ordering dependencies are described in [Codd 1970].

A final point about CODASYL data independence deficiencies concerns the existence of data base keys as unique identifiers of record occurrences. Most vendors of CODASYL-based DBMS have implemented the data base key so that its format has physical connotations in order for the system to be able to retrieve a specified record quickly. Even if the data base key is implemented as a logical identifier, the implicit relationship between key values and specific areas will make it difficult to reorganize the data base without affecting programs [Engles 1971].

On the positive side, an important contribution to data independence was made by DBTG in its distinction between schemas and subschemas [CODASYL 1971]. Effectively the subschema provides a subset of data base users with their own view of the data they require. These subschemas are mapped to the schema, and it is these mappings that help insulate the subschemas from many changes to the data model. More recently, the ANSI/X3/SPARC Study Group has recommended that the data model (schema) be replaced by two new schemas referred to as the Internal Data Base Schema and Conceptual Data Base Schema, in the hope that an additional layer of software will provide better data independence [Bachman 1975].

A look at hierarchical DBMS and their interfaces reveals that they too have deficiencies in attaining a wide degree of data independence.

Generally hierarchical systems do not provide a great deal of flexibility for structuring data and the ability to have very different subschemas is not present [Tsichritzis and Lochovsky 1976].

The relational approach to data base management systems as first outlined by Codd [Codd 1970] has been described as superior to the network and hierarchical DBMS models in the area of data independence. Physical data independence is said to be guaranteed as the completely flexible storage of relations can accomodate any changes in the storage structure. As for logical data independence, there are cases where data newly added to the data model may require changes in the accessing applications. In the case that the addition of data yields a relation that is unnormalized [Codd 1971a, 1971b], it would be necessary to reorganize some of the existing relations so as to maintain normalized ones. Relation reorganization can be seen to require reprogramming in [Michaels et. al. 1976]. Of the types of DBMS examined, the relational system is clearly advantageous in that it permits a wider degree of data independence. Again, though, we stress that time constraints may get in the way of the data independence offered by relational systems.

9.3 Content-Addressability

Implementors of DBMS who have focused on efficiency in data retrieval as well as update, have run into some problems within the context of conventional machine architecture. To cut down on the timely sequential searching through every record in response to certain retrieval requests, implementors have employed such aids as ordered lists, pointers, and inverted files. While improvements in retrieval efficiency are noticeable, those same aids are the source of performance degradation when updates are performed. In addition to the increased processing time involved to keep entries ordered and linked as updates are being made, there is an increased storage need in which the retrieval aids are to be maintained.

The use of associative memory or associative processors which allow data to be accessed by content instead of address location have been pointed to as a means of implementing more efficient DBMS than the ones currently available.

As for the level of technology in this area, Gertz [INFOTECH 1975] says that all of the associative memories presently available have two serious drawbacks. They are quite small by addressable memory standards, and they are very expensive because of the need for logic with every bit. Currently the work going on to develop and improve associative structures to be used with DBMS has focused on relational systems. (See [Ozkarahan et. al. 1976] for example.) Although all types of DBMS would appear to benefit from content addressable features, improved technology in this area will make direct implementations of the relational model feasible [INFOTECH 1975]. With the present difficulties in developing efficient relational DBMS, new technology in associative memories can certainly help establish the relational organization of DBMS as a viable alternative to the currently available commercial systems.

9.4 Data Sharing and Concurrent Updating Files

In a shared data base environment, there is a need to coordinate user access so that exclusive control over data areas may be attained without putting the data base system in an unrecoverable state of deadlock. Shared data bases, ones that allow simultaneous access by different users, are motivated by the necessity for efficient, responsive multiprogramming in the transaction environment [Chamberlin et. al. 1974].

In examining the problems of resource sharing in an operating system and trying to relate these problems to data sharing in a DBMS, we see that sharing is substantially more complex in the case of the DBMS.

First of all, in order to effectively promote maximized concurrency of access among processes, a DBMS should permit the basic lockable

resource (one that may be accessed exclusively) to be a very small unit such as an individual record. With essentially millions of such units, the order of magnitude of resource sharing and coordination is one that has not been seen in the context of operating systems. Another distinguishing feature of data sharing in a DBMS is the non-unique way that a user can request which data is to be locked. For example, it is not clear whether a request to lock the records of all red-haired employees and a request to lock those of all blue-eyed employees are in conflict. Other sharing problems particular to DBMS such as non-static resource categories and interdependent locks are described in [Chamberlin et. al. 1974].

As for particular design specifications, the DBTG proposal states that there is no provision for a user (through his interface) to selectively lock record occurrences and make them unavailable to concurrent users. The feeling for this was that such a feature would lead to a deterioration of performance [CODASYL 1971]. Instead, a warning mechanism is used to coordinate concurrent record updating by which a user is advised through an error-code whether a record has been altered in between the time it was accessed and the modification attempt was made. The specifications do, however, permit the programmer to request exclusive access on the area level, an area being a named subdivision of addressable storage space which may contain occurrences of records, owner-coupled sets, or parts of owner-coupled sets of various types. Though an area may contain more than the user's exclusive data needs, it makes the problem of deadlock simpler. A simple deadlock avoidance strategy is achieved by requiring a programmer to request all areas in which exclusive access is required at once (in one command). This way, a user will either get exclusive access to the areas requested or wait until other users free the already locked areas. Without permitting a user to lock an area after having already locked other areas, deadlock becomes impossible. As it is generally reasonable for a programmer to know ahead of time the areas he will need, this approach is workable. On a lower level, say

record occurrences, a similar deadlock avoidance approach would not be feasible as it would require a programmer to ask for all the records that are needed at one time [INFOTECH 1975].

The alternative to deadlock prevention is deadlock detection. Once the system detects deadlock, there must be some policy as to which user may have its processing undone or rolled back, and some procedure to accomplish roll-back to a point before conflicting exclusive requirements caused deadlock. In [Macri 1976] a design for deadlock detection and resolution is presented.

Systems that can perform deadlock detection and resolution will be able to provide more flexibility to the user when accessing the data base by permitting the users to lock different levels of data aggregates, and not requiring that all of the exclusive data items be stated at one time. Systems of this nature will incur the additional processing of deadlock detection and resolution procedures, but they will improve concurrency among users.

The trend in data sharing will be for systems to have interfaces which allow users to access data exclusively at various levels of granularity to promote improved concurrency of user access.

9.5 Distributed Processors

In a conventional DBMS, all of the major software components -- operating system, DBMS, and application programs -- execute on a single machine which has direct access to the data base on secondary storage. In the back-end system, the data base management function is implemented on a separate machine which has exclusive access to the data base [Canaday et. al. 1974]. In this way, the application program's requests are transmitted from the host machine to the back-end processor which itself may be a general purpose computer or a special purpose processor. In return, the back-end will deliver responses to the calling program with

associated status information.

Advantages of organizing a data base system in this way have been identified [Canaday et. al. 1974] as providing improvements in the following areas:

- data sharing -- The centralization of the data management function as provided by the back-end processor can help coordinate update requests from different users on computer systems which may be physically separated as well as being different makes or models.
- data base protection -- In terms of data integrity and security, two machines are better suited for detecting an error situation as a result of hardware or software failure than one machine attempting to do self-analysis. By centralizing the data base management features, user access can be controlled better.
- extensibility to other host machines -- With a back-end system already developed for a particular host machine, the job of writing the interface for the new host becomes a much smaller effort than developing the total data base management function on a new machine. This capability can be extended to smaller host-machines (minicomputers, for example) as a result of reduced core and CPU requirements when interfacing with a back-end processor.

As for the disadvantages of such a system, there is the added cost of a second processor, the additional response time overhead, and the potential unbalanced load on the multiple processors. XDMS, an experimental implementation of a back-end system by Bell Labs [Canaday et. al. 1974], supports the conclusion that data base systems of this type are feasible but that there are tradeoffs.

For the future and the impact systems of this type might have on the interfaces to DBMS, we feel that the one-record-at-a-time traversal of the DBMS permitted by the DML for CODASYL-type systems will be too inefficient. An interface that permits manipulation of larger data

aggregates would be better suited to reduce the data transmission overhead.

9.6 Distributed Data Bases

Whereas back-end systems permit the geographic distribution of processors accessing a data base, distributed data bases permit the geographic distribution of data so that different processors may access the data at varying locations. Advantages of distributed data bases have been shown [INFOTECH 1975] in cases where:

1) the total volume of data, or the total processing power required is simply too large for one system

2) a very fast response time is required at a number of geographically separate sites but with some need to communicate between the sites. Examples occur in command and control applications and production plant control systems.

3) the human organizations involved may require local autonomy, rather than central control.

4) an organization is faced with making the best use of a variety of existing systems

5) high reliability is required for the system as a whole, in the face of component failures.

A problem posed by distributing the data base is how to accomodate a user who needs data from more than one data site. One of the philosophies on how to permit data base accesses of this nature consists of requiring the user to specify a file transfer from one system to another when all of the needed information is not in one system. By a file transfer, the user is to some degree shielded from the conventions of the remote system. As this set-up requires user knowledge of the source of a file, we can extend this approach to the system taking care of file transfers. In that case, the system would access remote files as if they existed locally, without any explicit instructions from the user. Integrating

data base systems is particularly a problem when different DBMS involve different data models, data definition languages, data manipulation languages, and data formats. Some related issues that are still unresolved include finding ways to achieve transparency, provide translations, record statistics, maintain integrity, and control the system globally [Berg 1975].

Another problem that requires examination is the extent to which data redundancy can be permitted. Although data duplication from one site to another can improve the response time [INFOTECH 1975], the synchronization of the multiple data copies must be maintained. Attention should also be given to the issues of data privacy, especially if the potential for breaking the system rises as its complexity increases [Fry and Sibley 1976]. To relate advances in this area directly to the user interface, a user must be able to maipulate groups of records (not just singular records) so that data transmission times can be reduced.

9.7 Conclusion

We have described three ways in which DBMS use is related to HOL. First, both the self-contained and host dependent DBMS interfaces are embedded in programming languages to give a programmer full power over the data base. Specific implementations may favor specific programming languages by offering interfaces designed for those languages; however, DBMS that are aiming for a wide degree of use will be capable of interacting with a variety of host languages.

Second, the decreased programmer dependence on HOL as self-contained interfaces will be used for certain applications that would otherwise have been coded directly in an HOL. Efficiency problems related to these interfaces can reduce their effectiveness. In spite of efficiency concerns, the relational DBMS and its self-contained interfaces provide a wide degree of data independence. Improvements in

technology, particularly in the ability to use content addressability features for the implementation of relational DBMS, will cause relational systems to have a big impact on the commercially available DBMS.

Third, we examined specific requirements of technological advancements and their effects on the DBMS user interface and therefore on the programming language which contains the interface. We saw that as the need for effective multiple user access of a DBMS continues to grow, systems will provide flexible interfaces which permit the locking of various levels of data for varying amounts of time. With continued improvements in the ability to monitor and resolve resulting deadlocks, more systems will offer this type of flexibility. A final point about future DBMS and their interfaces was made by examining distributed processors (or back-end DBMS) as well as distributed data bases. Both designs have desirable features and as more work is done in these areas, we will see more data bases relying on these designs. Systems of this type should have interfaces that permit the manipulation of groups of records rather than singular records so that data transmission times can be reduced.

10.0 IMPACT OF DATA BASE MANAGEMENT SYSTEMS (DBMS) ON HOL PORTABILITY

One major virtue of HOL standardization is that it facilitates software portability. If this property is not to be lost when working within a data base management system (DBMS) environment, the DBMS-HOL interface must also be standardized. In addition, a convenient method must exist for transfer of the data base itself. The DOD has found that the cost of data transfer can exceed that of rewriting all systems and applications programs [Sibley and Taylor 1973].

The problem of data transfer is intimately coupled with that of data description. "The key to data compatibility is to develop a formal and explicit method of defining data precisely ..." [Yamaguchi and Merten 1974]. For these reasons, this chapter will concentrate primarily on the present and future trends in data description technology and its projected impact on HOLs. The emerging facilities for data manipulation are also discussed. However, it is important to emphasize the data description language (DDL) design, since decisions made here can affect the availability of certain facilities within the data manipulation language (DML).

Many of the details of data manipulation are dealt with in Section 9, and the reader is directed to it for the proper background.

At present, the problems of portability within the DBMS context have not been solved. "An application system based on a DBMS can be a degree of magnitude more complex and difficult to transplant than what can be called a 'conventional' system" [Olle 1974]. The solution to many of these problems has been seen to rest in the insulation of the user from the physical representation of the data (see Section 9.2).

This important concept of insulating the user from his environment will be seen to be a central theme of data description. Much work has been done on and continues to be devoted to defining the interfaces between the user and his data [ANSI/X3/SPARC 1975, CODASYL 1971, CODASYL 1973]. Only through careful definition and standardization of these interfaces can software portability be ensured [ANSI/X3/SPARC 1975].

A major effort is now proceeding towards DBMS standardization. The work of CODASYL's Data Base Task Group (DBTG) was begun in 1968. They have since proposed a standard DDL and DML extensions to COBOL [CODASYL 1971 and 1973]. Their proposals, based on the network model, have generated a large volume of discussion [Douque and Nijssen 1975, Lucking 1974, Michaels et. al., 1976].

This chapter deals with the facilities as proposed by CODASYL. Although there have been other facilities described or used [Boyce and Chamberlin 1973, Tsichritzis and Lochosky 1976], those of CODASYL's are stressed since theirs are being considered by many as a basis for an industry standard [Douque and Nijssen 1975].

10.1 The Data Dictionary

The first step in defining the data base is the specification of the names and meanings for all data attributes.* This can be accomplished through the use of a data dictionary. "The data dictionary provides the means of broadcasting definitions to the user community. It provides a narrative explanation of the meaning of the data names, its format, etc., thus giving the user a precise definition of terms" [Fry and Sibley 1976].

This precise definition is required to prevent several users from using the same name for different attributes or different names for the same attribute. While it will be seen later that the users may define their own views of the data base, a common set of names must be agreed upon for these definitions.

---

*A data attribute is any item, group of items or relationship between items in the data base.

The data dictionary has been envisioned as performing many additional tasks (e.g., maintenance of security, support of data validation and keeping of statistics on data base use [Plagman and Altshuler 1972]), but we will limit our discussion to those involving attribute definition. As we will shortly see, these other capabilities form part of the CODASYL schema DDL. The data dictionary can be seen as providing support for human understanding of the contents of the data base. In order to ensure that this understanding does not conflict with that of the applications programs, some interface must exist between the data dictionary and the DBMS.

This interface causes the data dictionary to have a very small effect on the HOL (e.g., through requiring compatible format definitions). However, as we will shortly see, the ability to rename data items through the sub-schema DDL makes this effect minimal. In fact, the knowledge of the attribute names can be considered the responsibility of the data base administrator (DBA), not of the applications programmer.

While the schema, sub-schema DDLs and the DBA remove the need for access to the data dictionary by the applications programmer, it is still required for the layman user. (Layman users are those with no programming background and little time or desire to acquire one.) In addition, the layman user is seen to interact with the system so infrequently as to retain little or no knowledge of it from one session to another [CODASYL 1971]. These users require some easily read, possibly interactive explanation of the contents of the data base. This will probably emerge as the responsibility of the data dictionary.

Another, analogous facility will emerge for use by the DBA. As the data schema become more and more complex, the DBA will require automated report facilities for their maintenance. For further discussions of schema report generation, see [Johnson 1975].

The data dictionary is also important in systems such as the one described in [Gerritsen 1975]. This system uses the definitions contained in the data dictionary and a set of queries to generate the data schema automatically. While this work is still in its early phases, it is indicative of the software tools that will emerge in the next 10 years to support the DBA.

10.2 The CODASYL DBTG Proposal

In his article on the CODASYL DDL, Waghorn states, "there can be on the whole little doubt that for the next 10 years most data bases will be CODASYL data bases" [Waghorn 1975]. This prediction, while open to some debate, has a great deal of evidence to support it. Many manufactures, seeking some immediate standard, have already produced CODASYL implementations. These include DBMSs for almost every large scale computer (e.g., Cullinane Corporation's IDMS for IBM systems, Digital Equipment's DBMS/10, Univac's DMS/1100 and Honeywell's IDS/II). One can also point to papers written by proponents of other data models (i.e., non-network) describing how these models may best be integrated with CODASYL's proposals (e.g., [Senko 1975], [Tsichritzis 1975]).

With this in mind, one is led to ask, "how far is the CODASYL DDL from becoming an official standard?" Let us first note that CODASYL considers their report "but a first step in the development of a common data description language" [CODASYL 1973]. The next step, the implementation and use of CODASYL systems,is now moving well along. Keeping this in mind and considering the general timescale of HOL standardization as discussed in Section 4, we foresee the acceptance of the final standard as three to six years off.

CODASYL divides the definition of the data base into three distinct phases. The logical structure and accessing paths are defined in the schema DDL. By an accessing path, we simply mean the information which must be supplied to the system in order to retrieve a given record. The schema

are restricted and reorganized by the sub-schema DDL specification. The sub-schema yield a particular view of the data base for use by a specific applications program. The third level of description, used to map the logical structure as specified in the schema to a particular storage device, is accomplished in the device and media control language (DMCL).

Before describing the actual languages which CODASYL has proposed, let us examine how this three-level approach ideally supports portability. We will consider the movement of three possible software components: (1) several or all the application programs, (2) either some part of or the entire data base, and (3) the entire system (i.e., both the data base and programs). By envisioning the availability of a CODASYL DBMS on both the source and target machine we can ignore the problems associated with DBMS transfer.

Given case (1) where only the programs are to be moved (i.e., a different database already exists or will be created on the target machine) we need simply modify our sub-schema definitions so that they coincide with the new system's schema. This assumes that all required data items either exist or can be derived from the new database. If new items need be created, we could incorporate them into the target schema without affecting existing programs.

To transplant the entire system, ideally no work is required. If internal data representation differs, the DBMS would access the information, recognize the need for translation (between DMCL storage definitions and DBMS memory descriptions), and perform the necessary conversion. However, due to new storage device utilization or large conversion overhead this could prove impossible or simply cost-prohibitive. Instead, one could write new DMCL definitions and supply these along with the old ones to some system which would automatically translate the database. This possibility is further explored in the automated data translation section. The transport of the data base alone can now be envisioned in light of the above discussion.

To realize the preceding ideals the three levels of data representation must be truly independent. For example, if an applications program is permitted knowledge of the physical representation of the data, then transport would necessitate program modification. We will now examine the CODASYL proposals with this in mind.

10.2.1 The Schema DDL

The schema DDL is the primary tool used by the DBA in defining his data base. The schema permits the DBA to specify all the data items that are to be part of the data base. These items are then grouped into records, and, finally, the records are related to each other in terms of owner-coupled sets (see preceding chapter for description of owner-coupled sets).

In addition the schema DDL provides facilities for: the specification of the access path to be used (its location mode), restricting use of the schema and the data it describes (privacy locks), and ordering members of the owner-coupled sets (sort keys) [Lucking 1975].

One very powerful feature of the CODASYL schema DDL is the ability to specify source and result data items. Source data items allow the inclusion of a data item within one record which actually resides in a different record, thereby eliminating the problem of data redundancy. Result data items are derived from information contained in the data base by data base procedures.

Data base procedures are actually capable of performing several functions. They can be used for data base security, computing result items, data validation and data conversion. While it has not been specified what language these procedures will be written in, it has been specified that the entire application program's environment should be available to it [Lucking 1974, Waghorn 1975]. If this access is to be supported, some facility must exist in the HOL to accomplish it. Such features are present in several HOLs (e.g., Simula's not hidden variable, CS-4's access statements and capability clause).

While CODASYL has included both data base procedures and virtual data items in its specifications, few implementations support the former and none the latter [Taylor and Frank 1976]. Consider what this omission means in terms of portability. It implies that even if a data item required by a transplanted program may be derived from its new data base, it cannot be provided. This forces either unnecessary modification of the program or the introduction of redundant data in the schema.

We have already discussed the ability to define access paths in the schema. This facility is very important, inasmuch as it provides efficient access to the data base. However, their specification in the schema requires the application program to supply the proper key when retrieving a record. This limits both the DBA's ability to reorganize the data base and the portability of programs. It also forces the application programmer to have knowledge of the underlying data schema.

10.2.2 The Sub-schema DDL

The sub-schema DDL, as opposed to the schema DDL, is language dependent. This dependence is limited to the description of record content. Among its language-independent features are the redefinition of privacy locks, and the abilities to declare alternate names for data-entities and to copy selectively, with a few minor changes, the descriptions of records [Lucking 1974].

CODASYL has defined their present sub-schema DDL to be used in conjunction with COBOL. Several of its features are directly derived from that language (e.g., level numbers, occurs clause, and picture clause). Research is now being conducted on the definition of a sub-schema DDL for FORTRAN [Stacey 1974, CODASYL 1973].

The problems with the COBOL sub-schema DDL rest not in its capabilities but in its inabilities. In the sub-schema DDL, CODASYL has left out the ability to define source and result items. This can be seen to preclude our ideal solution to the movement of programs (i.e., modification of only the sub-schema definition), unless source and target data items are grouped identically into records.

10.2.3 The DMCL

The device and media control language has not been formally defined by CODASYL. It is meant to be used to specify machine-dependent characteristics of the data base in a machine-independent manner. Most of the work in this area has been conducted in relation to automated data base translation, which we will now discuss.

10.3 Automated Data Translation

In parallel with the work of the Data Base Task Group, another effort has been going on at CODASYL. "The Stored Data Definition and Translation Task Group (SDDTTG) was formed ... to address a current problem ...: the lack of data transferability or the incompatibility of data. The purpose of the SDDTTG is to develop a method for concisely defining commonly used existing storage structures" [Sibley and Taylor 1973]. For reports on their work, see [SIGFIDET 1970, SIGFIDET 1972]. A formalized definition of the problems involved in system translation is presented in [Yamaguchi and Merten 1974].

There have been two major theses in this area [Smith 1971, Taylor 1971]. The first describes three languages for use in translating data bases. Two of these languages closely parallel the Data Base Task Group's schema DDL and DMCL. The third language, termed a criterion production system, is used to map the source definitions (made in the first two languages) into the target structures. The statements in this criterion production system resemble assembly language macro calls. As such, they are difficult to read and write but are highly flexible [Sibley and Taylor 1973].

Another approach has been taken by Housel, Lum and Shu in their development of An Interactive Migration System (AIMS). They feel that "requiring the user to redefine their data base and storage structures in terms of a complex language for the purpose of conversion makes the procedure difficult to be accepted by users in practice" [Housel et. al. 1974]. Their system, instead of requiring DDL specifications, extracts as much information as possible from

source applications programs and the source environment. It then interactively attempts to complete its picture of the necessary conversions.

At the University of Michigan, the ISDOS group has been responsible for a great amount of research in data translation [Taylor 1971, Sibley and Taylor 1970, Fry et. al. 1972, DTP 1973]. They have developed the Prototype system. The Prototype Data Translator has been used for the translation of COBOL data files created on Honeywell system to a DBMS, NIPS, format [Fry et. al. 1972]. The translation language is similar to Smith's criterion system in concept but is high-level in design. While this system is somewhat limited in its present power, it provides a test-bed for further research [Merten and Fry 1974].

After the acceptance of CODASYL as a standard for data description, many CODASYL-based systems will emerge for data translation. Within the next 10 years, such systems will make the problems of data base transport manageable. This will have an effect not only in relation to DBMS, but in the portability of all computer system. "We envisage that in the not-too-distant future every stored file will be accompanied by a tag (probably machine-readable) that describes the storage of the stored data. ... We further foresee a machine-readable DDL driving a generalized write routine for storing data onto different storage media" [Smith 1974].

10.4 CODASYL and the Relational Model

The fact that CODASYL has defined their data description languages with the network model in mind does not preclude their use with a relational interface. We use the term "interface" because it is indicative of the system resulting from such a combination. The user deals with his data base in a relational manner, while the underlying structure is that of a network.

In [Nijssen 1974], "instant guidelines" are presented for the DBA who wishes to support a relational interface to a CODASYL data base. These guidelines include: (1) limitations on the specification of search keys, (2) not sorting member records of owner-coupled set, and (3) the explicit definition

of access paths in the DML. In addition, Nijssen suggests seven modifications to the schema DDL that would assure its "peaceful coexistence" with the relational model.

These modifications to some extent limit direct access to the data base. The authors concur with the opinion that, "if a DBMS is to serve a wide user community, it cannot ignore those users who may need an access level close to the storage structure" [Michaels et. al. 1976]. What is required is the ability to permit certain users to operate at such a level while not requiring others to do so.

The major difference between the relational and network models lies in the inability of a network to support directly many-to-many relationships. These can be supported indirectly, however. For example, consider the problem of relating students to courses. A network provides no method for supporting this relationship directly, since we cannot simultaneously have a student record be the owner of several course records and a single course record own a set of student records. What is required is some intermediate record (e.g., grade) to maintain the proper linkages. If these links can be hidden from the applications programmer, then the goal of supporting a relational interface to a CODASYL data base can be achieved.

Some persons believe that the ordering of records should not be supported in a relational environment. Although ordering forms no part of the formal relational model, it can be useful for efficient accessing. Order is a very important element of the optimizations of relational queries discussed in [Smith and Chang 1975].

When the final CODASYL standard is approved, it will be capable of transparently supporting a relational interface. This will very possibly require additional work to be done by the HOL system that hosts a relational DML. The HOL compiler or run-time system will be responsible for optimizing the relational queries and specifying any intermediate linkages that are hidden from the HOL programmer. Such language facilities will probably begin to emerge in the next five to 10 years for two reasons: the relational model provides a higher-level mechanism for interacting with the data base [Michaels et. al. 1976] and promotes data independence (see preceding chapter).

10.5 Summary

A comprehensive HOL standardization program must address the problems involved in providing a DBMS interface. The reasons for this become obvious when we examine the portability of present-day software written within a DBMS environment. An order of magnitude increase in the cost for movement of such software is unacceptable.

The standardization of this interface is being carried out at present. The CODASYL DBTG proposals will emerge as the industry standard in the next six years. Therefore, any HOL selected as a DoD standard must include DML facilities and a sub-schema DDL in its specification.

This does not preclude the use of a relational model for defining the DML. For reasons of efficiency, however, the abilities for dealing with the data base on a lower level cannot be abandoned. Also, a relational DML requires additional support from the HOL system for hiding the underlying network structure of CODASYL's schema DDL.

Another area where the HOL must provide support for a CODASYL DBMS is that of data base procedures. The HOL must contain some mechanism by which the DBMS can access the program's environment.

The need for automatic data base translation is critical if system transport is to be economical. Although work in this field does not have a direct impact on HOL standardization, the ability to transport application systems is greatly affected by it. Within the next 10 years, commercial data base translation systems utilizing the CODASYL DDL descriptions will begin to become available.

In short, the movement of software within a DBMS environment presents a twofold problem. The first half, the standardization of the HOL-DBMS interface, is well on its way to solution. In the next four to eight years, standards will emerge for both data description and manipulation languages. The second half of the problem, data transport, will remain unsolved for eight to twelve years. Its solution will come in the form of automated data translation systems that input the old data base, both the source and target data definitions, and easily-written mappings between the two. The resulting output will be the new data base.

B. Operating Systems

11.0 THE IMPACT OF OPERATING SYSTEMS TECHNOLOGY ON HOLs

11.1 Introduction

As computer hardware technology has improved over the past several decades, the need for more powerful operating systems to exploit the processing capabilities of the machines has increased. A great deal of work has been done both in developing specific algorithms to be used in operating systems (e.g., memory management, scheduling) and in developing methodologies for designing, structuring, and implementing operating systems.

There are two important ways that operating system technology can have an impact on high-order language standardization. The first way involves the actual implementation of the operating system and the language in which it is done. Originally, operating systems were written almost exclusively in machine or assembly language. As the desirability of using high-order languages in other application areas increased and disciplines such as structured programming became prominent, interest in using high-order languages to implement operating systems also increased. Research has and is currently being done on what features to include in languages for implementing operating systems. This is discussed in the next section of this chapter.

The second way in which operating system technology can impact high-order language standardization involves the interface between the operating system and user programs. Work is being done in specifying standard operating system interfaces for high-order languages so that user programs can communicate with the operating system in an operating system-independent, machine-independent manner. Such an interface will clearly aid in the production of portable software. A related issue is that of providing a job control/command language to enable run-time parameters (e.g., specific file names) to be passed to programs. These issues will be discussed in the third section of this chapter.

11.2 Implementing Operating Systems

11.2.1 Synchronization Primitives

An important facility needed to implement operating systems is the ability to provide mutual exclusion and synchronization among processes. At the hardware level there can exist an instruction of the "test and set" variety which allows a flag to be tested and set in a single uninterruptable machine instruction. Such a facility is sufficient for implementing mutual exclusion and synchronization. The problem with "test and set" is that "busy waiting" is necessary if a process is unable to proceed.

A synchronization primitive to eliminate "busy waiting" has been introduced [Dijkstra 1968a]. This primitive consists of an object known as a semaphore and two operations known as P and V. This primitive is adequate for implementing mutual exclusion and synchronization without "busy waiting". If a process issues a P operation on a semaphore that causes the value of the semaphore to go negative, that process is blocked by putting it on a queue of waiting processes until another process issues a V operation on that semaphore. This scheme not only has the advantage of eliminating "busy waiting", it also gives the operating system (i.e., the implementation of the P and V operations) control over the order in which blocked processes are reactivated. It should be noted that the P and V operations themselves need to be implemented as critical sections. This, however, can be accomplished using the hardware "test and set" operation since the P and V operations are relatively short and conflicts to enter them would be minimal.

Several operating systems have been implemented using semaphores. Notable among these are the "THE" system [Dijkstra 1968b] and the Venus system [Liskov 1972]. The ALGOL 68 language includes semaphores as a built-in mode and has the operations "down" and "up" which correspond to the P and V operations [van Wijngaarden et. al. 1975].

Although semaphores with P and V are adequate for mutual exclusion

and synchronization, they are still fairly low-level primitives and are not entirely satisfactory. When semaphores are used for mutual exclusion because several processes are sharing a data structure, each process is responsible for ensuring mutual exclusion by the appropriate sequence of P and V operations and each process has direct access to all of the shared data structures. This can be extremely error-prone.

To overcome the problems of semaphores, a higher level primitive known as a monitor has been proposed by Brinch Hansen and Hoare [Hoare 1974a] (a similar concept known as a secretary has been proposed in [Dijkstra 1971]). A monitor consists of data structures to be shared among several processes and the procedures which have to be used to access them. Monitors are similar to abstract data type definitions that occur in several languages (e.g., SIMULA 67, CLU, ALPHARD, CS-4) but with important restrictions. Only a single procedure of a monitor may be active at any moment to ensure that several processes can not access shared data at the same time. Consequently, if one process is executing a monitor procedure and another process attempts to call a procedure of that monitor, the second process must be prevented from entering the monitor until the first process has finished. Thus, the (virtual) machine on which the processes run must handle the short-term scheduling of simultaneous monitor calls. The monitor procedures also have the ability to control medium-term scheduling of calling processes by delaying them in a queue (if for example a requested resource is not available) and then continuing their execution at some later time. When a monitor delays the calling process, that monitor is then free to receive a call from another process.

As mentioned before, monitors overcome the disadvantages of low-level primitives such as semaphores. Although slightly less efficient, perhaps, than low-level primitives, monitors enable complex synchronization problems to be solved in an understandable and reliable fashion.

Monitors have been included as part of the Concurrent PASCAL language [Brinch Hansen 1975a]. This language has been used thus far to implement the Solo Operating System [Brinch Hansen 1975b] for the PDP 11/45. Further details of Concurrent PASCAL and the Solo Operating System are contained in Section 11.2.4 A high-level primitive known as a "facility" which is extremely similar in nature to a monitor was used in the design of the SUE Operating System [Sevcik et. al. 1972].

Over the next decade, high-level primitives such as monitors will be used increasingly for implementing operating systems. The flexibility, simplicity, reliability, and abstracting capabilities of such primities make them extremely suitable for the expression of synchronization algorithms in a high-order language and far outweigh the possible, but slight, loss of efficiency from their implementation.

11.2.2 Structuring Concepts

With the advent of "structured programming" [Dijkstra 1972a], much attention has been given to the problem of suitably structuring large systems in order to reduce their complexity. In particular, work has been done in determining how to structure operating systems appropriately. Among the earliest work in this regard was Dijkstra's "THE" system [Dijkstra 1968b]. This system consists of levels of software in which each level defines a virtual machine to the levels above it. That is, each level uses features from the levels below it, abstracts out irrelevant details from those lower levels, and provides new features to the levels above it. This technique not only aids in the design and implementation of operating systems, but in their validation as well. For example, in the "THE" system, level 0 of the software handled real-time clock interrupts and implemented processes. Once this level had been validated, software at higher levels could be designed, implemented, and validated as sequential processes and the existence of the real-time clock could be ignored completely.

A more recent development in the design of operating systems is the family of operating systems concept [DeTreville and Tersoff 1976, Parnas 1976]. This concept consists of designing an operating system family using hierarchical decomposition. The resulting design consists of various levels with a number of modules at each level. Various members of the family are achieved by substituting in different but compatible modules. A module is generally substituted for another module for one of two reasons: 1) to change the algorithm(s) used in the module; 2) to change the implementation of the algorithm(s) used in the module.

A development in high-order language design that can support these structuring concepts is data abstraction. Data abstraction is discussed in detail in the chapter on "High-order Language Technology." Data abstraction originally appeared in the "class" mechanism in SIMULA 67 [Dahl et. al. 1970]. More recently, CS-4 [Brosgol et. al. 1975], CLU [Liskov 1976], and ALPHARD [Wulf 1974a] have included data abstraction facilities. The Concurrent PASCAL Language [Brinch Hansen 1975a] has included both the "class" mechanism of SIMULA 67 and the monitor mechanism (described in the previous section) which provides data abstraction facilities on data structures shared among concurrent processes. The Army Center for Tactical Computer Sciences currently has an on-going project [DeMillo 1975] which is investigating SIMULA-based constructs (in particular classes) as part of an effort to build usable languages for tactical control systems.

Over the next decade operating systems will increasingly be designed using structured approaches such as the family of operating systems concept. As the research in data abstractions progresses, these systems will be implemented in languages which provide such facilities.

11.2.3 Algorithms

There is currently a great deal of research being done on particular algorithms to be used in operating systems. Areas in which such research

is being done include storage management (e.g., paging algorithms) and processor allocation (e.g., scheduling algorithms). Although significant developments may be made in these areas in the future, they will not have any impact on high-order language standardization. As forecasted in the last section, operating systems will be structured in such a fashion that should a new algorithm be developed, a module containing it can be written to replace a module in the existing system.

11.2.4 Languages for Implementing Operating Systems

As already discussed in the previous sections of this chapter, the major language features that will be needed in the future for implementing operating systems are high-level synchronization primitives and data abstraction facilities. The language that currently comes the closest to providing adequate facilities is the Concurrent PASCAL Language [Brinch Hansen 1975a]. This language has been designed as an extension to the PASCAL language [Jensen and Wirth 1975]. In addition to the features present in PASCAL, Concurrent PASCAL contains concurrent processes, classes (as in SIMULA 67), and monitors. The language was developed for the PDP 11/45 at the California Institute of Technology. It has since been distributed to approximately 100 universities, 90 companies, and 45 research institutes. At approximately 40 of these institutions, plans are being made to move it to other computers. At least one operating system, SOLO [Brinch Hansen 1975b] has already been developed using Concurrent PASCAL. "SOLO is the first major example of a hierarchical concurrent program implemented in terms of abstract data types (classes, monitors, and processes) with compile-time control of most access rights." [Brinch Hansen 1975b]. Less than 4% of the system is written in machine code; the remainder is written in either Concurrent PASCAL or (sequential) PASCAL.

Over the next few years the Concurrent PASCAL language will increasingly be used to write operating systems, based on its already wide-spread

distribution. Only after the language has been used extensively and numerous systems written in it, can a proper evaluation of it be made. At that point either the language in its present form or with some modifications based on user experience will gain wide-spread acceptance for operating system development.

11.3 Interfacing User Programs with Operating Systems

11.3.1 Standard Operating System Interfaces

The second important way in which operating system technology can impact high-order language standardization involves the interface between the operating system and user programs. One aspect of this involves the facilities that the language itself provides to enable user programs to interface with the operating system. The approach taken in ALGOL 60 was to avoid the specification of any interface constructs (including input/output constructs) in the definition of the language. This approach leaves each implementation free to define its own constructs and greatly impedes transportability.

A much more reasonable approach is to have a standard operating system interface defined as part of the language. Such an approach was taken to a limited extent in the XPL language.

> By design, XPL object programs do not communicate directly with the operating system when requesting services. This interface is relegated to a submonitor program.... Although the XPL system was originally designed to operate under OS/360, it can run under any System/360 operating system with changes only in the submonitor. Changes in the operating system require only changes in the submonitor and not the recompilation of existing XPL programs [McKeeman et. al. 1970].

This approach was expanded in the design of the CS-4 language. "The purpose of the operating system interface is to standardize the interface between CS-4 programs and the variety of target machine operating systems which provide common functions in non-standardized forms. In general, if a particular capability exists in some form in the target machine operating

system, it is accessed via the corresponding OSI function; if the target machine operating system lacks some capability, the corresponding OSI functions are undefined for the implementation" [Brosgol et. al. 1975]. Alternatively, the capability could be provided by the OSI itself.

Over the next few years, further research will be done with regard to the standard operating system interface approach. The main question to be resolved is exactly what facilities should be provided for and what form they should take. Although CS-4 was designed using this approach, the designers are not completely satisfied with the particular interface used and feel that more work is needed in this regard. After this work has been completed and a standard operating system interface adopted, operating systems themselves could begin to provide their facilities in standard form so that by the end of a decade the need for the interface would decrease.

11.3.2 Job Control/Command Language

Another aspect of user programs interfacing with the operating system involves the (batch) job control language and/or (interactive) command language provided. Basically, the function of a job control/command language is to invoke programs and pass them parameters. For example, a given program might be written which needs to operate on a file. The system should enable the program to be written and compiled so that the particular file on which it is to operate on any given run can be specified as a parameter in the job control/command language.

An ANSI/SPARC/OSCL ad hoc study group has been established to investigate the question of a standard operating system command language. A fundamental question that needs to be resolved is whether to provide the full facilities of a general purpose programming language or whether a small, special-purpose language is sufficient.

In the SOLO Operating System [Brinch Hansen 1975b], the programming language provided to the users is (sequential) PASCAL. It was decided

that PASCAL could serve as the job control language as well [Brinch Hansen 1975c].

Over the next few years, further investigation and debate will occur over the question of a standard job control/command language. This question, however, is far less important than the other issues discussed in this chapter and will have no impact on HOL standardization.

11.4 Summary

Over the next decade, there are two areas in which technological developments in operating systems can impact high-order language standardization. The first area involves the language used to implement operating systems. The forecast in this area is that a language such as Concurrent PASCAL that contains high level synchronization primitives (monitors) and data structuring facilities (classes) will gain acceptance as a standard language for writing operating systems.

The second area involves the interface between user programs and the operating system. The forecast in this area is that a standard set of operating system interface features will be defined. Initially, operating systems interface routines will be needed to make non-standard functions available in standard form. Eventually, operating systems will provide their facilities in a standard form. In addition, a standardized job control/command language (very likely a general purpose programming language) for invoking programs and passing them parameters will be established, but this will have no impact on HOL standardization.

C. Real-Time Systems

12.0 REAL-TIME LANGUAGE EVALUATION

12.1 Introduction

"Real-time systems" is an all-encompassing term that includes systems from many diverse application areas. A common property of these systems is that they contain processing that must be completed under critical time constraints. In general these time constraints are relatively short (e.g., less than a minute), thus precluding such systems as payroll programs with critical but large time constraints.

The language requirements for real-time systems differ more in degree than in nature from the requirements for other systems. For example, the following five categories of language requirements apply to real-time systems as well as many other systems: reliability, maintainability, power, efficiency, and portability. The difference with real-time systems, however, is in the relative importance of the requirements, the degree to which each has to be satisfied, and the consequences of not satisfying a requirement.

In the past, the major emphasis for real time systems was placed on efficiency and this was used to justify the exclusive use of assembly language for real-time systems. The deficiencies of assembly language, however, are apparent when it comes to meeting the other requirements, especially reliability. With hardware getting faster and compilation techniques getting better, the use of high-order languages for real-time systems is gaining acceptance. In fact, given that the critical time constraints can be met, reliability becomes a much more important requirement than efficiency.

Section 12.2 discusses each of the five categories of requirements at a fairly general level and describes specific requirements in each of the areas.

Section 12.3 then selects those requirements which differ significantly in degree for real-time systems. For each requirement, language capabilities

and features to meet the requirement are discussed.

Section 12.4 contains evaluations of ten languages for their suitability for real-time systems. The requirements against which these languages are evaluated are the ones which differ significantly in degree for real-time systems (i.e., those discussed in Section 12.3). It should be noted that the languages are evaluated from their defining documents, and particular implementations are not considered.

Section 12.5 contains a summary of the language evaluations.

Discussing real-time language requirements and capabilities is not a straightforward task. It is not possible to state, for example, that a given capability is an absolute requirement for real-time and languages not possessing that capability are unsuitable for real-time processing (i.e., it may be possible to realize the capability in another way). Thus, the requirements and capabilities discussed in Section 12.3 are those that are considered extremely desirable for real-time processing but need not be considered absolutely essential.

It should be emphasized that the requirements and capabilities discussed are not necessarily the most important ones for a real-time language, bur rather the ones that differ significantly from other applications. For example, language masterability is an extremely important requirement in the category of reliability but no more so for real-time systems than for other systems. Consequently it is not included in the section on real-time specific requirements.

12.2 General Requirements for Real-Time Languages

12.2.1 Reliability

Reliability is an especially important requirement for real-time systems because of the nature of many real-time applications (e.g., process control, weapon systems). As Brian Randell has noted, "Reliability is not an add-on feature." Consequently, requirements must be placed on programming languages to support the production of reliable software.

12.2.1.1 Masterability

One important requirement is language masterability. The programming language is a tool with which the programmer solves the problem at hand. Thus, his full effort should be devoted to understanding the problem to be solved and not to understanding the tool [Hoare 1974b]. Properties of a language that contribute to its masterability are simplicity, naturalness, uniformity of features, implementation independence, completeness of definition, and the availability of effective documentation.

12.2.1.2 Understandability

Another important requirement is that the language support the writing of understandable programs. Understandability is important to reliability throughout the entire life-cycle of the software system. Programs that are clear and understandable are easier to debug, validate, and maintain. All the language properties that contribute to masterability also contribute to understandability. In addition, structuring features including functional and data abstraction also contribute to understandability.

12.2.1.3 Error Handling

Another important requirement for reliability is error prevention and detection. Language capabilities that support this are strong typing, useful redundancy, and explicitness (e.g., lack of defaults and implicit conversions).

Related to this requirement is the handling of exceptional conditions. As Hoare has pointed out, "It is quite unacceptable for a real-time program in the middle of its operation to suddenly give up, however polite the excuse" [Hoare 1975]. Consequently, facilities must be provided in the language to enable all errors and exceptional conditions to be handled by the program.

12.2.1.4 Implementation Correctness

A final reliability requirement is that the language implementation be correct. All errors should be due to programming errors at the source language level and not due to incorrect machine code generated by the compiler.

12.2.2 Maintainability

Maintainability refers to the ability to make changes to a program, both to fix errors and to support changing requirements of the program. In order to be able to make changes to a program, it is obviously necessary to understand the program. Thus all the language capabilities that support understandability, as previously discussed, also are applicable to maintainability. Understandability is especially important since frequently the person responsible for maintaining the program is different from the person who originally wrote it.

In addition, structuring capabilities that support modularity and locality are important for maintainability. A problem with making changes to a program is the need to ensure that the change does precisely what it is intended to do without affecting other portions of the system. The better structured and localized a program is, the smaller the probability that a change will adversely affect unrelated portions of the system.

It should be noted that although maintainability is a recognized goal for programs, it is not one that is easily identified or quantified. Although language factors have been identified that can contribute or detract from maintainability, we cannot measure a program's maintainability or even state with certainty that one program is more maintainable than another.

12.2.3 Power

In order for a language to be suitable for real-time systems, it must be powerful enough to conveniently handle the processing requirements of the system.

12.2.3.1 Sequential Computational Ability

One class of processing requirements includes the ability to handle sequential computation. The language features needed to support this requirement include control structures and data structures such as those found in general purpose programming languages. Sometimes a case is made for some "real-time specific" data types such as fixed point, matrix, and vector. The need for these, however, is highly dependent on the application area and is not intrinsic to the real-time nature of the system.

12.2.3.2 Parallel Computational Ability

Another class of requirements includes parallel processing capabilities. Many real-time systems conceptually can be thought of as several tasks operating in parallel with some degree of interaction. Language features that support parallelism enable the expression of programs that implement the system to follow closely the conceptualization of the system. In addition to features that support *the existence of* parallel tasks, features are also needed to control their interaction reliably.

12.2.3.3 Machine-Dependent Capability

An important aspect of the language power required for real-time systems is support of full use of the system hardware. Past attempts at general purpose high-order language design, having machine-independence as a goal, have not incorporated machine-dependent capabilities as an integral and uniform part of the languages. The fact that these have not been provided, even though needed, is a major reason for the predominance of assembly-language software in real-time systems [Wirth 1976].

An important reason for providing machine-dependent capabilities is that in many real-time systems, specific hardware (such as peripheral devices) is used having features especially designed for the particular application (or class of applications). The programming language must provide a means for accessing and using these special features. Machine-

dependent capabilities are also a means of achieving efficiency in that programmers, having explicit control of storage layouts and instruction sequences, can take maximum advantage of the hardware.

12.2.4 Efficiency

Although it is generally agreed that real-time software must be efficient, there has been a great deal of debate as to the relative importance of efficiency when compared with the requirements for reliability and maintainability. An ideal solution is to (1) provide software that is efficient enough to meet the time and space constraints of doing the job, (2) strive toward reliability and maintainability as long as the essential efficiency is provided, and (3) provide additional efficiency only when it will not compromise the reliability and maintainability achieved in (2).

The problem facing the language designer is that the "essential level" of efficiency can vary greatly, depending on the application requirements and the available hardware. The best he can do is (1) provide features that contribute to efficiency (and not provide features contributing to inefficiency) whenever there is no conflict with the other language requirements, and (2) attempt to minimize the number of basically-subjective trade-offs made between these requirements.

Machine-dependent capabilities can be used to achieve efficiency, as described above. In addition, a language can provide machine-independent "optimizer hooks", i.e., features that provide additional information that can be used by a "smart" compiler for optimization and that enable a programmer to specify time-space trade-offs. However, language features that require a smart compiler in order to achieve an efficient implementation should be avoided. Also to be avoided are features that have, independent of a compiler's cleverness, relatively high time or space costs and ones that require a non-trivial amount of run-time support.

12.2.5 Portability

There are at least two advantages in having computer software that can be transported from one set of hardware to another. First, capabilities

developed in one project can be transferred to another project, even with different equipment, and second, software lifetimes can be increased by carrying software from one machine generation to the next within the same installation [Tinman 1976]. The transportation of software not only has the potential for reduced costs, but also for increased reliability which would result naturally from the increased use of the same programs. There are several language requirements that are related to portability:

12.2.5.1 Machine and Implementation Independence

It must be possible to implement the language for a variety of target computers, and these implementations must be consistent with one another. Thus, the language design should avoid features that require the capabilities of any specific computer family for achieving a "reasonable" implementation. The specification of the language must explicitly indicate all places where machine and implementation dependencies are not avoided (such as the actual precision of floating-point numbers). It is also desirable to specify these dependencies in as independent (i.e., parameterized) a manner as possible. In other words, when dependencies cannot be realistically avoided, the goal should be to make such dependencies highly visible, so that their effect will not "cause any surprises" when a program is transported to another implementation.

12.2.5.2 Encapsulation of Machine-Dependencies

The previous section discussed the desirability of achieving, to the extent it is possible, machine-independence of language features. However, we have also noted that languages for real-time systems must provide the applications programmer with direct access to specific machine capabilities. Clearly the use of such capabilities will make real-time software machine-dependent. Just as machine-dependent portions of a language should be explicitly specified, so should machine-dependent portions of an applications program. Thus the language should provide a mechanism to insure that machine-dependent data and instructions are encapsulated for explicit

designation. When transporting a program, only those designated portions should need to be rewritten.

12.2.5.3 Re-Usability

Another goal often mentioned for real-time software is that programs developed for one system should be applicable to several other systems. Transportation of software from one system to another usually required that the program provide a "general-case" solution to the problem at hand. It would be desirable to have the necessary generality of the solution built into the program during its initial development, but achieving this is not a function of the language design. However, by requiring explicit interface definitions and features to support readability (as previously discussed), a language can at least assist in the development of re-usable software, and can aid potential re-users of the software, who must understand it in order to evaluate its applicability.

12.3 Specific Requirements for Real-Time Languages

12.3.1 Parallelism

The concept of parallelism is intrinsic to the nature of most real-time systems. This is the case because (1) various components of the system may have possibly-independent time-critical deadlines and (2) various peripheral devices may require immediate servicing.

Language features that support parallelism can not only contribute to the understandability and reliability of the programs produced, but can also facilitate actual parallel execution when a suitable hardware configuration is available.

Language features are needed to specify the creation (and termination) of parallel tasks and also the interaction among the parallel tasks. As Hoare has noted, however, "Methods of building parallelism into a high level language are a lively topic for ongoing research at the present time" [Hoare 1975]. Nonetheless, there are certain characteristics that the parallelism features must provide for real-time systems.

One important characteristic that must be present is that parallel

tasks be independent except for explicitly specified interactions. Thus, implicit inheritance of global variables by a parallel task is clearly inappropriate. Shared variables must be explicitly declared and accessed as such. Two possible mechanisms for this are critical regions and monitors. This characteristic is extremely important for reliability as well as understandability.

In addition to shared variables, features for synchronization among parallel processes are also required. The ability is needed for a process to delay itself if a particular condition is not currently satisfied. Similarly, the ability is needed for a process to indicate the fulfillment of a condition so that a delayed process may continue its execution.

Hardware interrupts from peripheral devices are conceptually a synchronization indication from a hardware process to a software process (i.e., an indication of the fulfillment of a condition). With a small amount of run-time support, this hardware-software synchronization can be unified with the *software-software synchronization facilities* of the language, thus eliminating the need for an explicit interrupt handling mechanism.

A final constraint that may be appropriate for real-time systems regards the creation and deletion of parallel process. Hoare feels that "The dynamic creation and deletion of processes is ill-advised. There should be a fixed number of processes; and when a process has nothing to do, it should simply wait for its next task" [Hoare 1975]. The question of dynamic vs. static creation of processes is really a time-space trade-off. Eliminating the ability to create processes dynamically in the language removes the system implementor's ability to make the trade-off himself. This is too restrictive for real-time systems.

12.3.2 Exception Handling

Exception-handling facilities are extremely important in a language for real-time systems. It is absolutely unacceptable for an error to cause a real-time system to lose control or terminate abnormally. The real-time

system itself should decide whether an error can be corrected adequately and if not provide an appropriate recovery action.

It should be noted that by exception handling, we are referring strictly to error handling. More general definitions of exceptions have been proposed [e.g., Goodenough 1975] that attempt to unify the mechanisms for dealing with errors and other "exceptional" conditions. Such an approach clearly includes the needed facilities for real-time error-handling but may provide more than is actually required for real-time systems.

Hardware interrupts can also be treated by the exception handling mechanism. However, there are conceptually different classes of hardware interrupts. As mentioned in the previous section, hardware interrupts from peripheral devices are conceptually a synchronization indication between parallel (asynchronous) tasks, and should be handled by the synchronization facilities of the language. Error interrupts (e.g., overflow) on the other hand are conceptually synchronous in that a given instruction stream has committed an error which causes invocation of an error handler

Language facilities are needed for specifying error-handling routines for all classes of errors that can occur. The mechanism must provide the handler with sufficient information (e.g., in the form of parameters) for the handler to diagnose the situation correctly. The handler should have the following options after completing its execution: (1) resume the routine that caused the error and allow it to retry the operation causing the error, (2) resume the routine that caused the error immediately after the operation causing the error, (3) abort the routine that caused the error and pass control to an appropriate recovery point. Cases (1) and (2) generally would be used when the handler is able to correct the error, whereas case (3) would be used when the error is uncorrectable.

12.3.3 Efficiency

Although efficiency should be an important consideration in any programming effort, it is especially crucial for real-time software. The

programmer must insure that the system (1) meets the critical time constraints of the application, (2) can be implemented within the size constraints of the hardware, and (3) has a time and space margin of safety so that system enhancements are possible during its period of use. The ease with which this can be achieved is significantly affected by the design of the programming language. Real-time languages should avoid features that contribute to inefficiency or that place a heavy burden on an implementation in order to achieve efficiency. Desirable features are ones which aid in the generation of efficient object code and ones which allow the program to resolve explicitly efficiency trade-offs such as time versus space.

12.3.3.1 Efficient Storage Allocation

An important design issue related to software efficiency is that of storage allocation strategies. Under a static strategy, the allocation and storage layout for all program data is determined prior to the program's execution. The same storage can be allocated, and used safely, for the local data belonging to two or more blocks or procedures whenever their activation periods are mutually disjoint. In addition, a static strategy implies minimal overhead on procedure invocations, and increased reliability since potential storage overflows can be detected at compile-time.

The disadvantage of static storage is that it is allocated throughout the execution, even when large portions may not be needed or used. Various dynamic allocation strategies alleviate this problem by allocating and deallocating storage as it is needed during the execution. For example, local procedure data can be allocated on a stack when a procedure is invoked and deallocated upon the procedure's return. The price for this more efficient use of memory is run-time storage management and more complex data accessing through a static chain or display. However, it is a price that must be paid if recursive procedures are desired in the language.

Other dynamic allocation strategies can be even more costly. For example, heap storage used for linked data structures is not automatically deallocated and requires periodic garbage collection. Another example is the requirement for cactus stacks when separate processes are permitted to reference the same automatic variables. General cactus stacks can add a great deal of run-time overhead since most hardware offers little direct support, although there are some notable exceptions [Cleary 1969]. Most real-time systems do not require the power of these dynamic strategies, and certainly cannot afford the additional costs.

12.3.3.2 Machine-Dependencies for Efficiency

A strong argument for assembly language programming is that it allows the program to take full advantage of the hardware capabilities. Thus the user has direct control over the efficiency of his data representations and instruction sequences. However, in recent years the ability of high-order language compilers to provide approximately the same level of efficiency has been demonstrated [Intermetrics 1975]. Features in these languages have been designed to reduce the need for machine-dependent constructs. For example, discriminated union and, on a more general level, block structure provide high-level, reliable means for achieving the efficiency of data overlays on the same physical storage.

In spite of these advances in high-order language design, there are still many cases where direct access to the machine is desired. For example, the program may be able to make assumptions that, for special cases, permit highly-specialized packing of data or optimization of control. While not generally condoning such practices (because of the potential impact on reliability, maintainability, and transportability), the language must permit them for applications having extremely critical time or space constraints.

12.3.3.3 Optimizer "Hooks" and User Controls for Efficiency

A language should not require the use of a smart (and therefore complex) compiler in order to generate efficient code. Often features can be incorporated into a language to provide a compiler with optimization information that the compiler itself could only obtain with great difficulty, if at all. For example, one can require explicit specification of numeric ranges, thereby allowing the compiler to determine efficient physical representations. Another example is requiring that the need for re-entrancy in a procedure be explicitly declared.

Other language features can be incorporated to give the user some control over his program's efficiency. For example, a machine-independent packing specification can be used to determine the time-space trade-off to be made between compact data representations and minimal data access times. Another example is the ability to specify whether the code for a procedure is to be closed and accessed from every invocation point, or openly regenerated at each of those points.

12.3.3.4 Avoiding Inefficiency

There are many features that, by their very nature, imply relatively high run-time costs and therefore should be avoided by the language designer. For example, dynamically-varying array sizes require costly storage allocation strategies as have been discussed previously. Even the requirement that the maximum size be fixed during the initial allocation does not eliminate the need for dope vectors and run-time address computations. Another area of run-time overhead is that involving checking. While desirable for reasons of reliability, the costs of such features can sometimes be prohibitive. A partial solution is to perform as many of the checks as possible at compile-time. This increases both efficiency and reliability at run-time, and in many cases can be accomplished without sacrificing necessary computational power. A third area to be avoided encompasses features that require extensive run-time support. A minimal amount of support must be provided for "administrative" purposes, especially

if the system includes parallel processes. However, extensive manipulation of files and peripheral devices, complex scheduling algorithms, sophisticated data management, and so on, are highly dependent upon the particular application. They are best "tailored" to meet the needs of a given system by that system's software and not by the supporting high-order language. However, this does not prevent the sharing of the same routine if it is applicable to more than one system.

12.3.4 Machine Independence and Machine-Dependent Capabilities

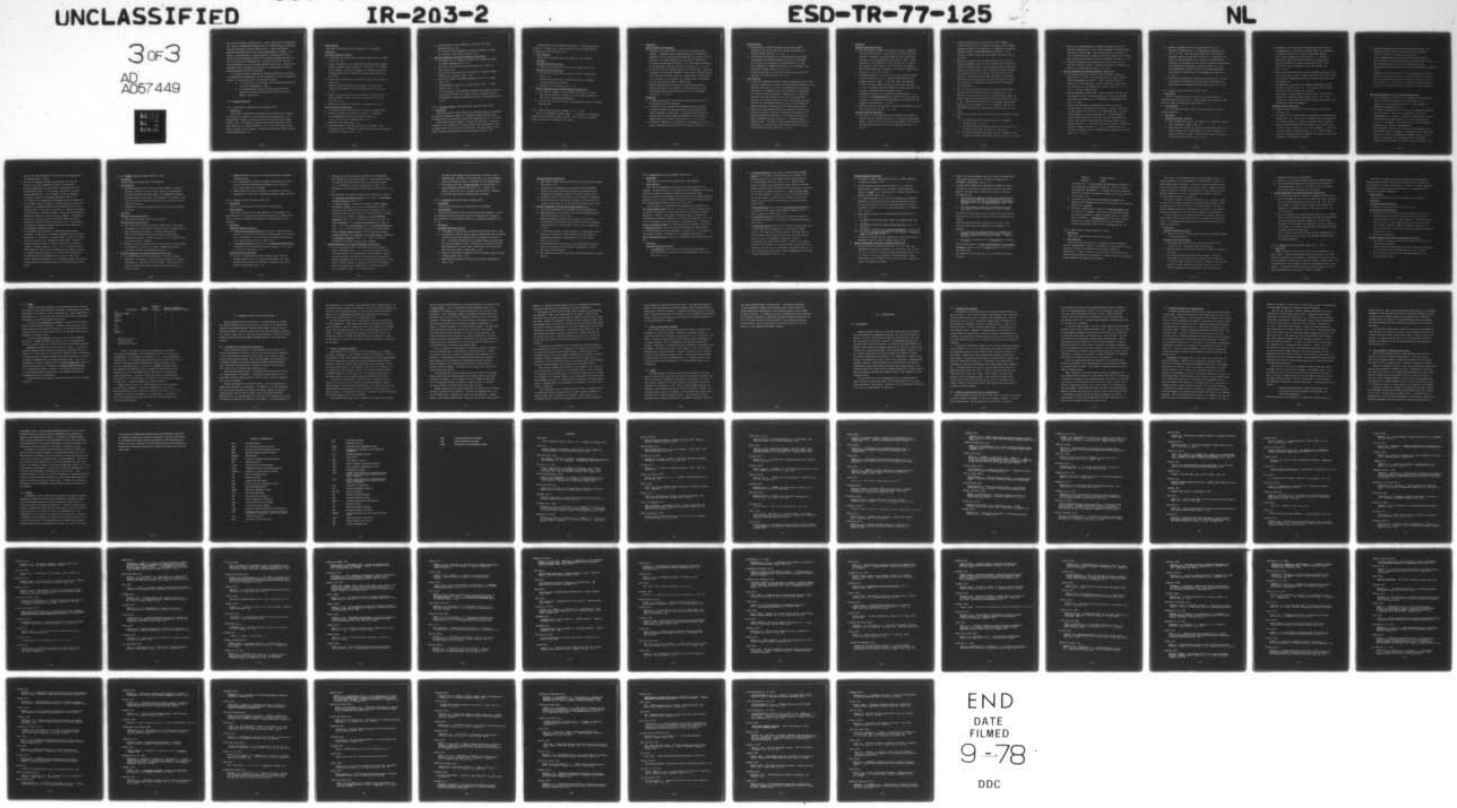
Although machine independence is a desired goal for real-time software, the widely divergent set of hardware designs makes its complete achievement virtually impossible. The best that can be done is to indicate explicitly all such dependencies as a part of the language specification. This includes both machine dependencies like integer range variations due to differing word sizes and implementation dependencies such as the number of allowable nesting levels. Often the dependency specification can be done using parameters in a uniform machine-independent manner.

Many real-time systems require the introduction of machine-dependencies so that the software can take advantage of specialized hardware. This is not only for reasons of efficiency, as we have discussed previously, but also because of application-dependent computational requirements such as the control of peripheral devices. Thus the supporting high-order language must provide machine-dependent capabilities. However, it should provide such capabilities in a machine-independent fashion that is consistent with the overall language design. For example, machine-dependent data types may vary from one implementation to another, but the rules for handling them should be consistent with those for other types. Similarly, direct code procedures should have a parameter mechanism that is both syntactically and semantically consistent (although not necessarily identical) with the mechanism for machine-independent procedures.

It is important that the language design makes the use of machine-dependent features explicit in an applications program. Machine-dependent

AD-A057 449 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2
HIGH-ORDER LANGUAGE TECHNOLOGY EVALUATION.(U)
OCT 76 T A DREISBACH, J L FELTY, I GREENBERG F19628-76-C-0225
UNCLASSIFIED IR-203-2 ESD-TR-77-125 NL

3 OF 3
AD A057449



END
DATE FILMED
9-78
DDC

data should be clearly labeled as such. Direct code should be encapsulated into explicitly-designated segments similar to BEGIN blocks or procedures (which may be either closed or opened inline). Again, the machine-dependent mechanisms should be as machine-independent as possible. For example, using parameters to machine-dependent procedures should not require that the programmer understand the general parameter storage layouts for a particular implementation. Nor should the rules constrain every implementation to use the same parameter organization. Instead, the programmer should be permitted to specify the layouts himself through a machine-dependent structuring mechanism.

By following the above approach of indicating explicitly all machine and implementation dependencies, program transportation can be accomplished reliably. The only constraint on the program is that:

> "(1) it violates none of the clearly defined resource constraints of either implementation, [and]
>
> (2) all program modules containing machine code have been replaced by modules having the same effect on the other machine" [Hoare 1975].

12.4 Language Evaluations

12.4.1 COL (Defining Document [Evans and Morgan 1976])

1. Parallelism

There are no facilities for creating parallel tasks in the language. If parallelism is needed it must be provided by subroutine call. The language does, however, provide facilities for mutual exclusion between parallel tasks. The language contains interlock variables which can be used either with a "region" statement or with "lock" and "unlock" statements to create critical regions. Interlock variables can also be used for synchronization [see p. 18].

2. Error Handling

There are no facilities for error handling in the language.

3. Efficiency

Optimizer Hooks/User Controls

a. A procedure or function may be declared as being open or closed [see p. 25, 79].

b. Arrays and structures can be specified as being packed, double packed, unpacked, double unpacked, or left alone. This enables the programmer to specify the degree of time/space tradeoff the compiler should make in choosing a storage representation [see pp. 36-38].

c. Assertions can be made for the compiler to use for optimization [see p. 94].

d. Compiler directives enable the programmer to instruct the compiler as to whether it should optimize time or space [see p. 73].

e. Arrays of structures can have the "parallel" attribute which indicates parallel rather than serial organization of the storage [see p. 38].

f. Ranges can be explicitly declared for integers [see p. 39].

Features Affecting Efficiency

a. The bounds of array indices must all be known at compile time.

b. Storage allocation may be static or dynamic. In addition, the programmer may explicitly allocate storage with "allocate" and "free" statements [see p. 29].

c. Recursion is permitted [see p. 25].

d. Procedures cannot access non-local dynamic variables. This scope restriction is intended to improve inefficiency by eliminating the need for a display [see p. 43].

e. Run-time support is only needed for "allocate" and "free" statements [see p. 96].

f. The "overlay" feature enables two structure elements to occupy the same part of the structure [see p. 36].

4. Machine-Independence and Machine-Dependent Capabilities

a. The default size of integer and logical (bit string) data items is hardware-dependent [see p. 87].

b. The size of any variable may be specified in bits, bytes, or words [see p. 28].

c. Any variables may be specified to reside at an absolute machine address or in a register [see p. 28].

d. Data may be grouped into machine-dependent units (e.g., pages) [see p. 24].

e. Machine-dependent code can be inserted using "code" ... "end" brackets [see p. 92].

f. Facilities exist in the language for making environment queries as to the word and byte size, default size for integers, logicals, and characters, number of hardware registers, and memory size [see p. 88].

12.4.2 Concurrent Pascal (Defining Document [Brinch Hansen 1975a])

1. Parallelism

Concurrent Pascal contains features for specifying parallel processes. All parallel processes, however, are created at system initialization. Processes cannot be dynamically created and terminated [see pp. 22, 25-26].

Shared data between processes is handled by "monitors". A monitor consists of data structures to be shared among several processes and the procedures that have to be used to access them. Monitors provide a high-level structures capability for handling shared data [see p. 38].

Synchronization is also handled by monitors. A monitor may delay a process in a "queue" until a given condition is met at which time the process may be continued [see p. 36].

2. Error Handling

There are no facilities for error handling in the language.

3. Efficiency

Optimizer Hooks/User Controls

a. Ranges can be explicitly declared for integers [see p. 10].

Features Affecting Efficiency

a. The bounds of array indices must all be known at compile-time [see p. 17].

b. Storage requirements can be determined entirely at compile-time and statically allocated.

c. Recursion is not permitted [see p. 34].

4. Machine-Independence and Machine-Dependent Capabilities

a. The range of integers, reals, integer case labels and integer sets, and the number of elements in a non-standard enumeration type are all implementation-dependent.

b. No facilities exist for defining machine-dependent data or including machine code.

12.4.3 CS-4 (Defining Document [Brosgol et. al. 1975])

CS-4 is a programming language system. It consists of a programming language and an operating system interface. This evaluation considers, without distinction, the capabilities of both.

1. Parallelism

Process Creation and Management

a. Processes in CS-4 are created by declaring variables of mode PROCESS. A SCHEDULE_PROCESS command associates a program-level procedure with the process and indicates that the process is to be scheduled. The actual scheduling of processes is determined by user-specified priority levels or, for time-critical processes, by user-specified real-time constraints [see Part II, p. 21, 25].

b. The storage class for PROCESS objects is not restricted. Therefore processes can be declared AUTOMATIC; these will be initiated dynamically (see the Efficiency section). Nevertheless, because there is no recursion in the language, the maximum number of processes that can be created is known at compile-time.

c. General capabilities are provided for process-state inquiry, process interruption and resumption, process termination, and guaranteeing that an executing process will not be preempted [see Part II, p. 21-29].

Shared Data

a. A SHARED storage-class is provided for data objects that are to be accessed by several processes.

b. Structured critical regions called UPDATE-blocks are provided to prevent disorder and deadlock which could result from concurrent accesses to shared data. All objects declared with the storage-class-attribute SHARED (PROTECTED) are only allowed to appear within such critical regions [see Part I, p. 20, 127].

c. A "corelink" capability is provided for the communication of data between processes having unrelated compilations [see Part II, p. 33].

Synchronization

a. Synchronization between processes is achieved by EVENT variables which can be SET by one process and WAITed on by other processes [see Part II, p. 31].

b. A user-specified connection can be established between EVENT and implementation-defined hardware conditions. The EVENTs will be SET on the occurrence of these conditions and thus allow responses to hardware interrupts to be programmed in the high-level language [see Part II, p. 32].

c. Process delays based on a real-time clock are provided by the TIME_WAIT functions [see Part II, p. 67].

2. Error Handling

a. A signalling mechanism provides a means of communicating the occurrence of an exceptional situation to user-specified handlers for that exception [see Part I, p. 151]. Signal-handlers may be passed information via parameters, and have rules which are consistent with those for normal procedures. Handlers may (1) handle the exception and return control to the point where the signal occurred, (2) decide the exception cannot be handled, take some other action, and transfer control to a point in the handler's immediate external environment, or (3) issue another signal.

b. Signals may be generated by explicit statements written in the program; in addition, they will automatically be generated for errors detected by the hardware (e.g., division by zero) or by compiler-supplied software checks (e.g., range and subscript bounds errors). The names of signals associated with errors are available to the user so that each program can define its own error-handling routines [see Part I, p. 265].

3. Efficiency

Optimizer Hooks/User Controls

a. The mode parameters RANGE (for integers and reals), PRECISION (for reals and fractions), and string-size can be used by the compiler to determine the minimum number of bits that must be allocated for the representation of an object's value. These parameters are not specified in terms of bits, but in terms of the (integer or real) range of the value space, the decimal digits of precision, and the number of characters, respectively [see Part I, p. 37, 44, 59, 100].

b. Procedures declared to be OPEN will have their bodies expanded inline at each point of call, thereby trading off space for reduced invocation times. In all other ways, the semantics of OPEN and normal CLOSED procedures are identical [see Part I, p. 148]. Procedures containing assembly or machine language code can also be declared OPEN and expanded inline.

c. A NORECALL attribute can be associated with procedures. It is similar to the reducible:directive of JOVIAL J73, and is used by a compiler in optimizing multiple procedure calls having the same argument values [see Part I, p. 150].

d. Checking-directives are available to disable the generation of run-time checking code [see Part I, p. 178].

Features Affecting Efficiency

a. The bounds of array subscripts are not required to be known at compile-time and therefore may require a "dope-vector" mechanism. However, the dope vector is kept relatively simple, since general non-contiguous cross-sections are not permitted [see Part I, p. 83, 87].

b. Storage allocation may be specified as either AUTOMATIC (within name scope levels), STATIC (at program level), SHARED (between processes at program level), or ABSOLUTE (for machine-dependent objects of mode MSTRUCTURE) [see Part I, p. 20]. Pointers and HEAP storage requiring garbage collection are not provided.

c. Recursive procedures are not permitted [see Part I, p. 140]. Therefore AUTOMATIC data (but only the dope vectors for dynamic arrays and strings) may be laid out statically with restricted scopes of access, and not require references through a display. The language specification uses the term "allocate" and "de-allocate" to indicate when the data's storage is dynamically made available or unavailable, and where routines are invoked for initiation (of values for data and states for processes) and termination.

d. The user can control the efficiency of initiate and terminate routines using the OPEN attribute as discussed above. In addition, object declarations using language-supplied modes can specify that the initiate routine is not to be invoked [see Part I, p. 18]. The terminate routines for the language-supplied modes perform no action [see Part I, p. 26]; therefore, they can be implemented as body-less OPEN procedures which require no overhead.

e. Efficiency of storage using overlaid data can be achieved in three ways:

   i. By compiler-determined storage layouts for AUTOMATIC data [see Part I, p. 20];

   ii. By compiler-determined storage layouts for objects of discriminated UNION mode [see Part I, p. 113]; and

   iii. By choosing storage-unit-values such that component fields in machine-dependent MSTRUCTURES overlap [see Part I, p. 183].

f. Many of the language-supplied procedures and operators will be invoked at compile-time if all of their arguments or operand values are known at compile-time. The resulting values can be used not only in contexts requiring compile-time values, but also in contexts that normally would require run-time invocations [see Part I, p. 173].

g. Although conversions are possible using explicit calls on construction routines, implicit conversions resulting in unknown overhead are not permitted [see Part I, p. 26, 28].

4. Machine Independence and Machine-Dependent Capabilities

a. The language definition reflects a strong attempt to make the specification of objects and the manipulation of their values as machine-independent as possible. This is exemplified by the description of the REAL mode, the REAL value space, the arithmetic operations, and the PORTABILITY constants [see Part I, p. 45, 267, 67, and 203, respectively].

b. Machine-dependencies whose effect cannot be hidden from the user are defined in terms of parameterized values which are available to the user as constants (e.g., MAX_MACHINE_ INTEGER_VALUE) [see Part I, p. 204].

c. Machine and implementation-dependent data types such as bit strings, bytes, and words are provided for. However, such types can only appear in invocations of the MSTRUCTURE mode, and therefore encapsulate the machine-dependencies. Machine-dependent storage layouts can also be specified using the MSTRUCTURE capability; this includes the positioning of fields relative to one another [see Part I, p. 183] and the absolute locating of MSTRUCTURE objects in memory [see Part I, p. 192]. A control-feature can be used to disallow the use of MSTRUCTURES and absolute allocations [see Part I, p. 175].

d. Assembly language code can be incorporated into a CS-4 program by encapsulating it in a MPROCEDURE declaration [see Part I, p. 193]. The syntax and semantics of these machine-dependent procedures (and their parameters) are consistent with those of normal procedures (e.g., they may be OPENed inline). The rules are consistent, but not identical because certain machine and implementation-dependencies cannot be hidden (e.g., the machine-dependent register location of an actual parameter is specifiable, and may be used in the assembly code [see Part I, p. 195]).

e. Implementation-dependent linkage conventions for calls to externally-specified procedures written in other languages can be established using EPROCEDURE declarations [see Part I, p. 199].

12.4.4 Coral 66 (Defining document [Coral 1970])

1. Parallelism

Coral 66 directly provides no facilities for task creation, synchronization, and data sharing. These capabilities are seen to rest within an undefined supervisory system [see p. 2].

2. Error Handling

No facilities exist within Coral 66 for the purpose of error or exception handling.

3. Efficiency

Optimizer Hooks/User Controls

a. The programmer must specify the number of bits required to represent a fixed-point number [see p. 9].

b. Table definition allows the packing of several data items into a single computer word. Packed data may be either signed or unsigned fixed-point elements. Their magnitude and precision are specified in terms of bits [see p. 11-12].

c. Overlaying of data items may be accomplished either manually or automatically. The overlay statement enables the programmer to explicitly declare the re-use of global data space in which common variables are stored [see p. 17]. The block structure of Coral 66 allows the compiler to automatically overlay variables with disjoint activations [see p. 17-18]. This feature may be overriden by use of the pre-set list [see p. 16-17].

d. Macro facilities are provided, so that repeated code may be inserted in-line [see p. 44-46]. These facilities can be used in place of procedures, thereby eliminating the overhead associated with a procedure call.

e. The sequence of evaluation for arithmetic expressions is left partially undefined [see p. 23]. This permits local optimizations which alter the order of evaluation.

f. Three diadic, logical operators (Differ, Union, and Mask) are defined for use between typed primaries [see p. 23]. This allows use of these commonly available machine instructions within higher-level language statements.

Features Affecting Efficiency

a. Dynamic storage allocation is not required on block entry or exit [see p. 3].

b. In normal use, no runtime checks are performed on array subscript bounds [see p. 3]. While this is desirable from the prospective of program efficiency, it certainly reduces program reliability.

c. Arrays and tables always must be declared to be of compile-time constant size and must be of at most two dimensions. In addition, no array slicing is supported [see p. 10-11]. These features limit the language flexibility somewhat, but cause array and table accessing to be extremely efficient.

d. Recursive procedures must be explicitly specified [see p. 32], allowing both more efficient calling and static allocation of local variables for non-recursive procedures.

e. Formal parameters must be fully specified and exactly match the attributes of the associated actual parameters [see p. 35-36]. Such conventions permit full static (compile-time) type-checking and eliminate the need for run-time conversions at procedure entry.

f. Variables may be initialized prior to program execution through the use of a pre-set list [see p. 14-16]. The pre-set list may also be used for defining constants, however, such constants are more difficult for the compiler to recognize than would be those declared in a language containing an explicit constant statement.

4. Machine-Independent and Machine-Dependent Capabilities

a. In table-element declarations, it is suggested but not required that the word BIT be permitted as an alternative to the comma in a location specifier [see p. 12]. This leads to an implementation-dependent syntax.

b. The internal representation of floating- and fixed-point numbers is left undefined. While this is necessary for efficient implementation on a variety of machines, it means that the results of the BITS and logical operators are machine-dependent [see p. 21-23]. Furthermore, no facilities exist in the language for determining the internal representations.

c. Code statements allow for the inclusion of non-standard instructions within Coral 66 programs. One suggested use of these statements is to permit the embedding of assembly language routines. While it is proposed that non-standard code be capable of accessing Coral variables, no formal interface is defined. The

ability for code statements to return a value is permitted but not required [see p. 28-29].

d. The Coral 66 defining document specifies that the type and scale of all actual parameters be known at compile-time. We have already seen that this leads to highly efficient run-time procedure calls. It is, however, restrictive in terms of procedure generality. For this reason, [Coral 66] suggests a syntax for optional non-standard parameters [see p. 37]. This leads to both implementation-dependent syntax and semantics.

e. In an attempt to permit support of Coral 66 on a wide variety of machines, the language definition allows subsetting. The major features seen to be most likely omitted are: recursive procedures, table facilities, fixed-point numbers, logical and BITS operators, overlaying facilities, and floating-point numbers. In addition, features not contained within the official definition may be approved for specific fields of work [see p. 56]. This lenient maintenance of language standards leads to an ever-growing set of non-compatible language dialects.

f. The programmer has the ability to reference absolute memory locations. In addition to the absolute communicator which permits assignment of a variable to a specific memory location, two operators are included in the language for this purpose. Location (i) yields the memory address of variable i and [j] is a reference (pointer) to memory location j [see p. 22]. To facilitate use of these operators, conventions are established for memory allocation of variables and arrays [see p. 14]. While these operators and conventions allow the programmer greater control, they limit certain compiler optimizations (e.g., efficient register utilization, storage allocation to minimize paging, and the overlaying of data space).

12.4.5 FORTRAN (Defining Document [BSR X3.9 1976])

1. Parallelism

FORTRAN offers no capabilities for parallelism.

2. Error Handling

a. FORTRAN allows the user to specify an ERR parameter to every I/O statement. When an I/O error occurs, control is transferred to the specified error processing code. The FORTRAN draft proposal standard does not require all I/O be trapped in this manner. Some I/O errors may still generate the system action, which is usually program termination.

b. FORTRAN has no language-defined facilities for handling arithmetic exceptions.

3. Efficiency

Optimizer Hooks/Usage Controls

a. There are no optimizer hooks or user controls.

Features Affecting Efficiency

a. The bounds of all arrays are fixed at compile time. All storage requirements can be determined at compile time, or certain storage may be dynamically allocated at procedure entrance time. [BSR X3.9 1976] explicitly specifies which variables may be dynamically allocated. Dynamic allocation is not required; there is no provision for recursion.

b. FORTRAN is a simple language with few options and no features which are expensive in run-time or storage. FORTRAN compiles into code which runs well on present day machines.

4. Machine-Independence and Machine-Dependent Capabilities

a. The FORTRAN definition of data precision is entirely machine dependent. For example, "A double precision datum is a processor approximation to the value of a real number. The precision, although not specified, must be greater than that of type real", [BSR X3.9 1976].

b. FORTRAN provides no facilities for defining machine or assembly language inserts.

c. Use of EQUIVALENCE to specify alternate definition of the same storage locations builds machine dependency into a program and is often used for that purpose.

d. FORTRAN offers no structured method of processing an internal representation of data nor of addressing explicit memory locations.

12.4.6 JOVIAL J3 (Defining Document [AFM 1967])

1. Parallelism

There are no facilities for parallelism in the language.

2. Error Handling

There are no facilities for error-handling in the language. The specification of the language does address the detection of compile-time errors; however, the detection and handling of run-time errors is not addressed.

3. Efficiency

Optimizer Hooks/User Controls

a. A basic:structure:specification can be used to indicate whether entries in a table are to be layed out consecutively or in parallel (e.g., consecutively for all first words, followed consecutively by all second words, and so on) [see p. 48].

b. A packing:specification can be used in ordinary:table:declarations to indicate to the compiler what packing strategy is to be used in laying out tables [see p. 49].

Features Affecting Efficiency

a. The size of arrays must be fixed at compile-time; since all indexing begins at zero, the bounds are therefore also fixed. Although table sizes can be variable, the maximum size must be fixed at compile-time [see p. 47].

b. The issue of storage allocation strategies is not addressed. The language does provide for global vs. local scoping of names, but does not allow the specification of local data as static (i.e., allocated only once at program level) [see p. 61].

c. Procedures and functions may not be called recursively [see p. 58, 59].

d. The programmer can explicitly specify the overlaying of data by an independent:overlay:declaration [see p. 42] or a subordinate: overlay:declaration [see p. 47].

e. In numeric:formulas, "necessary conversions between floating and fixed or integer are carried out automatically", thus hiding potential overhead from the user's awareness [see p. 20]. "Such conversions are also carried out where necessary in assigning the value as required in numeric: and dual:assignment: statements" [see p. 26]. Although the manual states that "the actual:parameters of a procedure:call:statement must agree in type ... with the formal:parameters of the procedure:declaration" [see p. 28], it later states that "there must be compatibility between formal:parameter:items and the corresponding actual: parameter:formulas and :variables, of the same nature as exhibited by assignment:statements" [see p. 29]. Thus, the implicit conversions can appear in any of these contexts.

4. Machine-Independence and Machine-Dependent Capabilities

a. One of the primary goals of Jovial was that the language be machine independent [see p. 2]. This goal has been achieved in that the language can be implemented on many machines. However, the most stringent requirement that the meaning of programs be machine-independent has not been met. The programmer must be aware of the "external effect" of his data in terms of its language-specified representation (although the actual representation may be different) [see p. 12]. The data types

provided by the language are all specified in terms of a bit-representation, and many of the operations (e.g., the arithmetic ones) are defined in terms of the effect on the representation.

b. The language provides a direct:statement for the insertion of machine/assembly language code [see p. 31]. However, the effect of such code is not isolated from the rest of a program, as could be done across a procedure-like interface.

12.4.7 JOVIAL/J3B (Defining Document [Softech 1976])

1. Parallelism

There are no facilities for parallelism in the language.

2. Error Handling

There are no facilities for error handling in the language. In fact, the language specification states, "Errors not detected by a compiler are considered to yield undefined affects in an executing program" [see p. 1-4].

3. Efficiency

Optimizer Hooks/User Controls

a. A procedure or function may be declared as reentrant [see p. 3-16].

b. A procedure or function may be given the INLINE attribute which means that the code for the procedure will be inserted at the point of call. The INLINE attribute may be declared globally in which case all calls are compiled inline. Alternately, the INLINE attribute may be declared in a local scope in which case only calls within that scope are compiled inline [see p. 3-21].

c. Items in a table may be declared to have no packing, medium packing, or dense packing [see p. 4-4].

d. Tables may be declared to have serial or parallel organization [see p. 4-8].

Features Affecting Efficiency

a. The bounds of table and array indices must all be known at compile time [see p. 4-7].

b. Storage local to procedures and functions have the AUTOMATIC attribute but may be allocated statically at program entry or dynamically at procedure (function) invocation [see p. 3-12].

c. The manual does not state whether or not recursion is permitted.

d. OVERLAY declarations enable the programmer to specify that two elements should start at the same storage location [see p. 4-18].

4. Machine-Independence and Machine-Dependent Capabilities

a. Some machine dependencies are parameterized [see Appendix A].

b. Machine-dependent data specifications can be encapuslated into "specified" tables [see p. 4-15].

c. Machine-dependencies permeate the language in the form of word-boundary and alignment restrictions of which the programmer must be aware (e.g., word-boundary restrictions in tables [see p. 4-9]).

d. A J3B single precision, double precision, or fixed point literal value "may not denote its true decimal value, but rather an implementation-dependent approximation to its true value" [see p. 1-13, 1-14].

e. Bit strings exist as a data type in the language and functions exist for obtaining the bit (or byte) representation of other data types in the language.

f. The language does not have facilities for machine/assembly language inserts.

12.4.8 JOVIAL J73/I (Defining Document [RADC 1976])

1. Parallelism

There are no facilities for parallelism in the language.

2. Error Handling

There are no comprehensive facilities for error-handling in the language. The specification of the language does not address the occurrence of errors and does not define the behavior resulting from errors. This is the case for both hardware errors such as overflow during arithmetic operations [see p. 4-6] and software errors such as out-of-bounds indices during table:entry referencing [see pp. 2-5 and 4-16].

The only means for user-defined error-handling is provided by the error:flag:statement:name [see pp. 1-12 and 1-15], which is used to signal an unrestricted transfer of control when errors occur during conversion operations. However, the decision to transfer control is based upon implementation-dependent variations in the size of values [see p. 1-15] and will not necessarily be uniform across all implementations. In addition, error:flag:statement:names can only be specified for explicit invocations of mapping:functions; they cannot be specified when the same functions are invoked implicitly during assignment, initialization (i.e., presetting), and parameter passage. The behavior for conversion errors when an error:flag:statement:name is nil is not addressed by the language specification.

3. Efficiency

Optimizer Hooks/User Controls

a. A size:specifier can be used to indicate the minimum number of bits that must be allocated for the representation of an object's value [see p. 2-1].

b. A structure:specifier can be used to indicate whether table:entrys are to be laid out consecutively or in parallel (i.e., consecutively for all first words followed consecutively by all second words, and so on) [see p. 2-6]. However, this feature is needed only because the language does not have more general record-of-array and array-of-array structures in addition to the array-of-records structure (i.e., table of table:items) that it does provide. In addition, the use of parallel tables can actually result in less efficient accesses when entries requiring multiple words are partitioned into non-contiguous storage areas.

c. A packing:specifier can be used in ordinary:table:declarations to indicate to the compiler what packing strategy is to be used in laying out tables [see p. 2-7].

d. An interface:directive can be used to inform the compiler of overlapping allocations to be accounted for during optimization [see p. 6-8].

e. A reducible:directive can be used to indicate that multiple calls with the same actual parameters to a function will always result in the same value for the function. Thus the compiler will be able to save the return value from the first call and use it in place of subsequent invocations. There is no guarantee that side effects having an impact on the return value will be detected, or that side effects resulting from the function call will occur at each point of invocation [see p. 6-9].

f. The language permits additional compiler directives to be defined by each implementation [see p. 6-2].

Features Affecting Efficiency

a. The bounds of table indices must be fixed to integer constants at compile-time [see p. 2-6].

b. Storage allocation may be either permanent (i.e., static at program level), temporary (i.e., automatic within procedure levels), or based (i.e., referenced with a pointer using machine-dependent address units) [see p. 1-6].

c. It is unclear whether or not recursive procedures are permitted. Therefore, it is not known whether temporary data may actually be laid out statically, but with restricted scopes of access, or whether it must truly be allocated dynamically and referenced via a display.

d. Efficiency of storage using overlaid data can be achieved in four ways:

   i. By compiler-determined storage layouts for temporary data [see p. 1-6];

   ii. By programmer-controlled layouts using based data [see p. 1-6 and 2-17];

   iii. By explicit specification via overlay:declarations, [see p. 1-8];

   iv. By choosing location: specifiers such that entries in specified: tables overlap [see p. 2-5].

e. The language provides implicit conversions, which could result in overhead of which the user is unaware [see p. 1-12].

4. Machine-Independence and Machine-Dependent Capabilities

a. The representation of data items [see p. 2-2, 5-7, 5-8], the addressing and relative positioning of data [see p. 1-6, 1-8, 2-5, 2-9, 4-17], and the operations for manipulating data [see p. 1-12, 4-1 through 4-15] are all extremely machine-dependent. These dependencies are not hidden from the user, but must be taken into account throughout his program.

b. Some of the machine-dependencies are in terms of parameterized values which are available to the user as constants (e.g., BITSINWORD, ADDRESSSIZE) [see p. 1-2].

c. Although machine-dependent parameters are accessible, they do not provide a user with the capability to control the actual physical representation, location, and layout of his data.

Two examples are:

i. The meaning of an absolute address is dependent upon the manner in which addresses are assigned for a particular operating system, but usually corresponds to the relative address from zero for the complete:program address space [p. 1-8].

ii. The actual number of memory units allocated to an item can be greater than those specified by the programmer [p. 2-3].

The second of these implies that the behavior of overlaid data, when used for free unions, may vary from one implementation to another.

d. There are many places where implementors are free to make decisions which affect the semantics of the language. For example:

i. Since floating-point representations are implementation-dependent, each implementation must specify the exact semantics associated with the use of these ... constructs [p. 2-2].

ii. The result of a branch into a loop:statement is undefined [p. 1-11].

The evaluation orders of actual:input:parameters, actual:output:parameters, [see p. 2-18], and formula operands [see p. 4-7] are unspecified.

The semantics of many operators are not completely specified. Thus for,

```
ITEM II S;          "signed integer"
ITEM FF F;          "float"
```

the result of S + F would probably be equivalent to FLOAT(S) + F, but could also be implemented as FLOAT(S + INT (F)) or even FLOAT(BITOF(S) and BITOF(F))! Although the type of the return value for arithmetic operations is defined, the size of the value is not [see p. 4-6].

e. The machine and implementation dependencies permeate most aspects of the language, and cannot be encapsulated into isolated portions of a program.

f. The Level I subset does not contain the direct:statement found in the full J73 language. The subset has no alternative means for providing machine or assembly language code as an integral part of a program. A linkage:directive can be used to specify variations in linkage conventions for calls to externally-specified procedures written in other languages [see p. 6-5].

12.4.9 PL/I (Defining Document [BSR X3.53 1975])

1. Parallelism

There are no facilities for parallelism in PL/I.

2. Error Handling

On the occurrance of any error or exception condition defined by the PL/I implementation, control is transferred to a user-defined routine (called an ON-unit) for processing that condition (called an ON-condition). When a condition arises for which no routine has been defined then the condition is processed by a system routine, which normally prints a message and terminates the program.

The routine for processing exception conditions may consist of a single statement (not an IF-statement) or a BEGIN block. Neither of these forms accepts a parameter. Routines to process conversion exceptions, arising when converting from character data to numeric, can use a built-in function to obtain the character or field causing the error, and this character or field may be modified to attempt a correction to the condition causing the exception.

For other arithmetic exception conditions (e.g., overflow, divide by zero) PL/I provides no method of determining which statement caused the exception nor what the inputs to the operation were. The only choices present for the PL/I programmer are to return to the statement causing the exception, or to GOTO another statement in his program.

The PL/I ON-unit is also used to process I/O exception conditions. Aside from the end-of-file condition, the I/O exception conditions are errors. The remarks concerning arithmetic exceptions apply equally to these I/O exceptions.

3. Efficiency

Optimizer Hooks/User Controls

a. Items in an array or structure may be declared aligned (unpacked) or unaligned (packed).

Features Affecting Efficiency

a. Arrays may have dynamic bounds, which need not be set until allocation time. Strings may have varying length.

b. Extreme care must be taken to avoid invoking implicit conversion routines.

c. PL/I requires a sophisticated run-time system. Storage is allocated in four different ways, three of which are dynamic. I/O may be stream (character oriented) or record (block oriented). Record I/O may be sequential or direct, keyed or not keyed. The storage explicitly allocated by the user must also be

restored to the pool of freed memory.

d. GOTO's out of a scope may require extensive bookkeeping work.

e. PL/I syntactically permits the specification of operations on aggregate data items (e.g., A+B where A and B are arrays) which can be easily optimized by the compiler.

4. Machine-Independence and Machine-Dependent Capabilities

a. PL/I has no provision for machine or assembly language inserts.

b. Use of the UNSPEC built-in function allows the user to work with the bit-string representation of any data type. The value of this feature is limited, however, because PL/I provides no capability for addressing explicit core locations (e.g., for I/O controllers).

c. The PL/I description of floating point, fixed point, or integer data is entirely machine independent, although some machines may not have the required hardware capability. The PL/I user specifies only the number of bits/digits of precision needed by his program. The compiler then generates the appropriate single or double precision instructions.

d. A PL/I user may build certain hidden machine dependencies into his program, using the overlay (PL/I DEFINED) capability and the pointer mechanism.

12.4.10 SIMULA 67 (Defining Document [Dahl et. al. 1970])

1. Parallelism

SIMULA 67 does not contain facilities for true parallelism. It does, however, contain quasi-parallel facilities [see pp. 67-71] that enable co-routines to be established. Any prefixed block has the potential of becoming a quasi-parallel object by issuing a "detach" [see p. 69]. A quasi-parallel object issuing a "detach" relinquishes control to its creator. A quasi-parallel object may explicitly relinquish control to another quasi-parallel object with a "resume" statement.

Facilities for mutual exclusion of shared data are unnecessary because a quasi-parallel routine retains control until it explicitly relinquishes it to another routine with a "resume" or "detach" statement. Synchronization among quasi-parallel routines is accomplished with "resume" and "detach" statements.

2. Error Handling

There are no facilities for error handling in the language.

3. Efficiency

Optimizer Hooks/User Controls

a. There are no optimizer hooks or user controls.

Features Affecting Effeciency

a. The bounds of array indices need not be known at compile-time.

b. Storage allocation is either automatic at block entry and exit or explicit using the "new" statement. An object allocated explicitly with a "new" statement can be deleted either when it is no longer referenced (implying a garbage collector) or on exit from the scope in which the class declaration for the object occurs [see p. 67].

c. Recursion is permitted.

4. Machine Indepencence and Machine-Dependent Capabilities

a. The range of integers and the range and precision of reals are implementation-dependent.

b. Character sets, collating sequences, and initialization of character variables are implementation-dependent [see p. 28-29].

c. No facilities exist for defining machine-dependent data or including machine code.

12.5 Summary

In order to evaluate languages for their suitability for real-time processing, we have identified four categories of requirements that differ significantly in degree for real-time applications. These categories of requirements are: parallelism; error handling; efficiency; and machine-independence and machine-dependent capabilities.

In terms of the parallelism requirements, only CS-4 and Concurrent PASCAL provide complete facilities. SIMULA 67 contains facilities for quasi-parallelism which permit co-routines. COL provides facilities for mutual exclusion and synchronization but not for the specification and management of parallel tasks.

In terms of error-handling, only CS-4 and PL/I provide adequate facilities. FORTRAN provides error handling only on I/O operations and J73/I provides error handling only on explicit conversion operations.

Efficiency requirements are much harder to evaluate. In terms of optimizer hooks and user controls for specifying trade-offs, COL, CS-4, CORAL 66, J3, J3/B, and J73/I all provide an adequate range of facilities. In terms of specific language features that affect efficiency all of the languages do reasonably well, with the exception of PL/I. COL, CS-4, and CORAL 66 provide an adequate level of machine-independence as well as providing facilities for the inclusion of machine dependent features in an encapsulated fashion. Although J3 and J3B provide facilities for encapsulating machine dependencies, the languages themselves are permeated with machine dependencies.

The following table concisely summarizes the results of the language evaluation.

	Parallelism	Error Handling	Features for Efficiency	Machine Independence Machine-Dependent Capability
COL	+		*	*
Concurrent PASCAL	*		+	
CORAL 66			*	*
CS-4	*	*	*	*
FORTRAN		+	+	
J3			*	+
J3B			*	+
J73/I		+	*	
PL/I		*		
SIMULA 67	+		+	

* adequate facilities

\+ some facilities

inadequate facilities

Overall, CS-4 comes the closest to satisfying all the real-time specific requirements. The one drawback to CS-4 is the lack of an implementation. This drawback also applies to COL. Of the implemented languages only PL/I provides error handling facilities but PL/I fails with regard to efficiency, machine independence, machine dependent capabilities, and parallelism. J3, J3B, J73/I, FORTRAN, and CORAL 66 provide adequate efficiency but fail with regard to parallelism and error handling, and J73/I and FORTRAN have no facilities for encapsulating machine dependencies. Concurrent PASCAL does well with parallelism as does SIMULA 67 to a lesser extent, but both fail with regard to efficiency, encapsulation of machine dependencies, and error handling.

Despite this critical look at the suitability of these languages for real-time processing, it should be remembered that no single language capability can be stated as an absolute requirement (i.e, it may be possible to realize the capability in another way). Consequently, none of the languages evaluated can be ruled as absolutely unsuitable for real-time processing.

13.0 HARDWARE SUPPORT OF HOLs FOR REAL-TIME

Real-time applications are receiving an increased amount of interest. The primary reason for this is that the cost of each unit of computational capacity has decreased markedly during the past decade, allowing uses of computer automation, monitoring, and control that were previously unfeasible. As a result, a great deal of attention is being devoted to achieving the best machine architecture and programming language design for real-time applications.

13.1 Characteristics of Real-Time Applications

Real-time applications are not characterized by any single set of special requirements or criteria. Many of the features that are desirable in real-time processing systems are also needed in an efficient, large, non-real-time computing system. Also, the development of time sharing as a computing strategy has required that large central computers possess most "real-time" characteristics, primarily the ability to deal rapidly with stimuli from an unpredictable, changing world.

Real-time applications tend to be dominated by sensing and/or control of a nondeterministic environment. Information about the world is not predictable before program execution, and the control responses necessary can not be pre-computed.

Another requirement in a real-time system is that the computational speed of the processor be predictable. For example, it is often necessary to know the time required to process an information sample, so that the actual sampling time of the next sample can be accurately determined. Additionally, it is also necessary to be certain of the maximum time required to execute the time critical routines in the system. If the performance of

the system were to vary greatly, much processing time, processor power, and thus money would be wasted in anticipation of the very unlikely maximum time. Finally, if the system performance is very critical and some unforeseen special set of circumstances developed that decreased the processing power of the computer facility, the system could fail.

Real-time applications vary in the frequency and time criticalness of their operations. The constraints placed on the machine architecture vary accordingly. Usually, though, when real-time computing is discussed, systems that are operating near the limit of machine capacity are of interest. A computer architecture that supports real-time, then, is an architecture that can process the maximum amount of information in the minimum amount of time, for a cost that is within the budgetary constraints of the system.

13.2 Ways to Increase Throughput

Many features that increase the raw data throughput of a computer have been incorporated in the traditional machine architecture. Real-time applications, though, often require the manipulating of many asynchronously-occurring events, such as inputs from several different sensor systems aboard an aircraft or through many remote data access terminals. Since the time relations between the external events can not be predicted, it is necessary tht the processor respond to the events through interrupt processing rather than by polling, even if the polling process were to use known properties of the external world to decrease inefficiency.

A system driven by external interrupts must be capable of acknowledging the interrupt with a minimum of architecture-induced overhead. General-purpose register-oriented machines have the advantage of reducing memory traffic by maximizing the use of registers to store temporary data, but when it is necessary to process an interrupt, saving the registers and restoring them later can be very costly.

Interrupting one process that is running and initiating an interrupt

service routine that determines the correct system action is referred to as a context switch. All of the processor status information that is not contained in the memory for the executing program must be saved for later restoration. Such information resides in dedicated registers within the processor itself and can include information related to any or all of the following: memory mapping, I/O device allocation, operations in progress using external devices (such as outboard arithmetic processors, I/O devices, or another processor in a multiprocessor system). The operations involving external devices can potentially finish while some context other than the initiating process is in control of the processor. Any machine architecture that maintains this type of information in machine registers that must be dedicated to the executing context must perform its context switching inefficiently and thus increase the overhead involved in processing each occurrence of an external event.

There is, however, a modification of the general register structure that can eliminate much of the overhead in context switches. Instead of having a single set of registers and machine status registers, it is possible to include many sets of registers within the single machine. A state register must be included to differentiate between the sets of program registers, but context switching becomes a simple matter of changing that one register. The Varian 520/i [Varian 1969] and the PDP-11/45 provide this ability.

Providing multiple sets of registers does not eliminate the problem, though, since there are usually many more processes required than there are sets of registers. Since, however, the most frequent context swap is to the operating system, providing one additional set of registers can cut the number of context swaps by at least one half.

Another alternative to providing many sets of registers is to maintain as small a machine state as possible. The earliest computers had a very small machine state -- typically only one or two accumulators, a program counter, and a machine status word. Saving the context was a simple matter.

However, to constrain current machines to such a rudimentary architecture would entail sacrificing the advantages of current technology.

When general register machines were first introduced, they possessed two advantages. First, the access speed of the general registers was much faster than the memory access time, so results could be saved in much less time. Second, the registers, because they are limited in number, can be addressed with a very few bits, thus making the instructions that access them shorter, saving on memory space. The first advantage has been eroded through the increased availability of high speed larger memories. The time difference between a register reference and a memory reference can now be made very small. The second advantage is still just as real as it ever was and is still a very strong motivation for general register machines.

An alternative to providing a set of registers in special memory in the processor unit itself is to incorporate the functions of the registers into the central memory of the computer system. Memory speeds have increased greatly, and overall memory performance has improved. Allowing registers to exist in memory permits the allocation of many sets of registers, one for each process, while still incurring only the overhead of changing processor register set pointers upon each context switch.

Most real-time systems also require a large amount of I/O. The size of each individual I/O transaction is frequently small -- just the amount of information read from one sensor or the control signals to one actuator. The frequency of I/O access, however, can be much higher than for the same scale computer operating in a batch-oriented information processing application. Hardware support for decreasing the overhead of this kind of I/O access is desirable.

As is mentioned in the I/O section of this report, incorporating the I/O access and control facilities into the memory mapping hardware can greatly reduce the I/O overhead of the system. Each process can be allocated its own unique set of input/output devices, sharing them with

other processes as required by the application. The operating system or resident real-time executive need only intervene when a process wishes to change its I/O device allocations. Although each process directly controls the I/O devices allocated to it, it remains a user (as opposed to supervisor) program. Thus the vulnerability of one process to incorrect operation of an unrelated process is reduced.

13.3 Impact on High-Order Languages

As is shown above, there are basically two means of increasing the throughput of a real-time system: reducing the overhead involved in switching contexts and increasing the efficiency of the I/O portion of the system. Neither of these features should have any effect whatsoever on a properly designed, machine independent, general purpose high-order language. The brevity of context switches is completely transparent to the programmer, the effect being only on the compiler to use the particular method supported by the hardware. Likewise, the form of the I/O should not affect the high-order language either. Current high-level language techniques mask the actual implementation of I/O transfers under a particular executive and would make the I/O through mapped memory also be transparent.

13.4 Summary

The same hardware techniques and machine architectures that make general computing efficient will make real-time processing efficient also. There are basically two means of increasing the throughput of a real-time system: reducing the overhead involved in switching contexts and increasing the efficiency of I/O. Both of these hardware features are desirable in any computer system, but they are especially valuable in a real-time system, due to the frequency of both input/output transactions and externally stimulated context changes. The overhead of context switching can be reduced by decreasing the number of processor registers

that must be altered upon a context switch. Input/output efficiency can be increased by allowing the processes direct control of the I/O devices without the intervention of the operating system on each access to verify validity. I/O overhead can also be decreased (and the reliability of the system increased) by separating unrelated processes through the use of an I/O structure such as that discussed in the Input/Output section of this report. None of these means of increasing efficiency have any effect on a well-designed high-order language.

14.0 COMMUNICATIONS

14.0 Introduction

Numerous research advances in the last several years have affected real-time systems and communication systems. For real-time systems in general, the major step forward has been the identification of failure-prone programming methods and the design of programming languages and techniques that avoid those methods. The language Concurrent PASCAL, for example, offers high-order language constructs to define the relations and interactions of parallel processes. The languages PASCAL, SIMULA 67, and CS-4 require strong data typing, which enables the compiler to assure only approved operations are performed. For communication systems in particular, the advances have come in three major categories: communications hardware, computer interface hardware for communications, and software techniques for organizing systems developed around communications. The hardware advances will make available improved service at a greatly reduced cost; the software advances will enable systems to take advantage of this improved service.

The real-time area is so encompassing that the forecasts made in other sections of this report apply to real-time systems as well as non-real-time systems. This section, therefore, will be restricted to advances in the area of communications in particular.

14.2 Communications Hardware

A new method of data transmission will become available within the next ten years. It will be the greatest single advance in the history of the now-twenty-year-old field of digital data communications. The new method involves the use of "lightwave" communications [Buchsbaum 1976], which offer the advantages of satellite communications -- high to very high band width -- without the major disadvantage: long delay and, so, poor response times. "Lightwave" communications will be suitable for a wide variety of applications, including interactive time-sharing, file transfer, and interactive graphics applications requiring high data rates for large volumes of data (e.g., new screen) as well as rapid interactive response.

A light wave transmission system is built from miniature semiconductor lasers [Kressel et. al. 1976], fiber optic wave guides [Chynoweth 1976], and integrated optical circuits [Conwell 1976] serving as switches and amplifiers. Although Bell Labs has developed lightwave communication only recently, the physical fiber bundle is already smaller, lighter, and much more rugged than copper wire. Furthermore, the transmission is unaffected by electrical disturbances that would disable radio communication. The fiber bundle also offers a data throughput some orders of magnitude greater than wire cable. In short, lightwave facilities will offer lower cost, lower error rates, fewer outages, and higher data transfer rates than present systems. Lower costs will perhaps lower tariffs and thus open new applications that are not cost-effective at current tariffs. High data rates will open applications requiring bulk transfers and quick response.

14.3 Computer Interface Hardware for Communications

The development of Large Scale Integrated (LSI) circuitry will continue to affect hardware development in the entire computer industry. For computer communications, two developments are of particular interest:

micro-computers and single-chip interfaces which are able to assume much of the message protocol control. For two important recent protocols, SDLC from IBM and HDLC from IWG6 of IFIP, single integrated circuits exist to control the transmission of the individual frames. The logical control (i.e., message acknowledgements, retransmission) must still be done by a processor.

In the last several years, the cost of micro-computers has decreased and their quality has increased [Economist 1976]. Many of them still have strange addressing modes and inconvenient instruction sets, though, but in restricted applications such as device or line controllers, the ease and flexibility of programming outweigh these disadvantages. One disadvantage that may now restrict the use of micro-computers in communications controllers results from the new models that appear frequently. New micro-computers are often vastly superior to old ones, so the old ones are withdrawn from the market and any systems using the old ones must be redesigned. Designers are reluctant to commit themselves to a component which may not be available for the life of the product.

Within two years micro-computer technology will reach the stage at which improvements in price/performance can be made without changing the external specifications. Mini-computer technology has been at that stage for several years.

Taken together, single-chip interfaces and micro-computers mean that communications interfaces of great power can be built at little cost. A network node computer having responsibilities like those of an ARPA IMP (but without any teletype and very few indicator lamps) could be built today (in reasonable quantity; i.e., 1000 or more) for a hardware unit cost under $5000. Mass production could lower this price substantially. However, accurate cost predictions cannot be made, because the division of responsibility among the various system components will change. A safe prediction is that system components with great capability at low cost will be available for specific applications.

14.4 Software Techniques for Communications

In computer communication the most widely used (not the most widely written about, however) configuration is the connection of a single terminal to a single channel on one particular machine, whether over a leased line or over the switched public network. The purpose of systems so configured is to save the cost and time of physically transporting information available at a remote site (e.g., a New York City sales office) to a computer center. Several refinements designed to further reduce costs have been built: for batch terminals, each of which is only intermittently active, multidrop configurations where several remote terminals use the same leased line and respond only when specifically addressed; for interactive terminals (e.g., teletypes on a time sharing system) data concentrators are used to multiplex many low speed devices onto a high speed line. With all these refinements, remote access is becoming an increasingly adequate substitute for immediate physical access to the computer center.

Networking, on the other hand, is more than just a substitute for physical proximity. ARPANET is the prototypical computer network, so most of the following discussions use ARPANET terminology. The ARPANET comprises a set of hosts, which are dissimilar and provide the computational service, and a "subnet" that guarantees correct delivery of messages or announcement of failure. The subnet decomposes a message into fixed-length blocks called "packets" each of which is individually routed to the destination where they are collected and reassembled into the original message before delivery to the destination host. (The subnet is sometimes called a "packet-switching network.") By allowing computers to communicate, networks provide a service that is qualitatively superior to that provided by the remote-access terminals described in the preceding paragraph. When the computers are dissimilar, as in ARPANET, the user may select the machine best suited to his application. However, even when a

network is composed of hosts all of the same type, the set of programs and the data bases available on different hosts are different.

Over the next few years, national telephone companies (or private companies) in several countries will offer packet-switched network facilities. The countries include the U. S. (Telenet), Canada (Datapac), the Nordic countries with a regional system, Great Britain (Experimental Packet Switching Service), and France (RCP & Transpac) [Rybczyski et. al. 1976]. Aside from these systems, a number of private systems are being developed for university research [Mills 1976], military applications and for the internal use of large firms (Boeing Company and National City Bank of New York). DEC (DECnet) and IBM (System Network Architecture = SNA) offer commercial systems with some networking capability. All of these systems are different. In some, the network leaves to the hosts all responsibility of maintaining data security; in others, the network guarantees error-free delivery. Some systems encourage the use of front-end processors (FEPs) between a host and the network; in other systems this has no particular benefit. Over the next several years, the performance of these different networks can be evaluated in actual practice. Experience will reveal the best network approach for a given specialized or generalized application.

This activity on the part of telephone companies, together with an increased activity on the part of equipment manufacturers, make it safe to predict that the use of network systems will increase in the years ahead. Initially, network systems will be used most frequently as lower-cost remote batch terminals. However, some persons will use networks and networking techniques to provide new services. Current research areas include:

- Network Operating Systems [Kimbleton and Mandell 1976]
- Distributed Data Bases [Kimbleton and Schneider 1975]
- Distributed Computation.

The major problem in this research is a common one: defining what the

system should do. What a Network Operating System should do in a network composed of identical hosts [Mills 1976] is probably different from what it should do in a network composed of dissimilar hosts. When data representations are different in different machines, then Distributed Data Base capability must use powerful data formatting and conversion techniques.

Over the next several years, research will continue in determining those features that should be included in the functional definitions of a network operating system, based on user needs. Features likely to be included are resource-sharing [Kann 1972], load-leveling [Mills 1976, Kimbleton and Schneider 1975], and distributed data bases [Kimbleton and Schneider 1975].

14.5 The Influence of Communications on HOLs

There are two important ways in which communications can have an impact on high-order language standardization. The first involves application languages needing to make use of communication facilities; the second involves languages used to implement communication systems.

Application languages needing to make use of communication facilities include such languages as FORTRAN and COBOL. Communications will have only negligible effect on such languages because operating system interfaces will shield application programs from all details of the I/O. The only effect of communications is likely to be a growing demand for real portability of these languages so that a given program with a given set of data will produce the same results on any machine in the network.

Languages suitable for implementing communication systems include such languages as Concurrent PASCAL [Brinch Hansen 1975a], CS-4 [Brosgol et. al. 1975], and COL [Evans and Morgan 1976]. In general, HOLs are unsuitable for doing the hardware-dependent, time-critical, character-by-character processing that was required for communications before the

development of LSI. Putting these processing functions in an LSI interface facilitates the use of an HOL for implementing the higher-level aspects of the communications system. For example, the node processor (IMP in ARPA terminology) could be programmed in an HOL, thus simplifying the implementation of the routing algorithms, the queueing strategy, and the special processing required for messages of different priority.

Development of the host operating systems is the area in which the use of an HOL can be most effective. The distributed nature of a network implies that components will be developed at different installations; an HOL can help assure the compatibility of these components. Use of an HOL allows the clear specification of parallel processes and their interaction. The "monitor" feature of Concurrent PASCAL (see Section 11.2.1) is a particularly good example of a high-level structuring mechanism that can facilitate the development of complex network applications. Other important features of an HOL for network implementation include portability, control and data structuring capabilities, strong type-checking especially across interfaces, and efficient object code. Portability is especially important in network development because dissimilar hosts can be part of the network.

14.6 Summary

Developments in both communications hardware and computer interface hardware for communications are making communication systems and applications developed around such systems increasingly attractive due to decreasing costs and increasing capabilities. Lightwave communications, which will become available within the next decade, will be the greatest single advance in the history of digital data communications. The development of LSI circuitry will enable protocol processing functions to be handled by micro-computers and single-chip interfaces which will facilitate the use of HOLs in the implementation of communication systems. Networks, particularly packet-switched networks, will offer a full range

of facilities for communicating among similar and dissimilar computers. For reasons of reliability, security, portability, and cost, the software for the network and network operating systems will most effectively be written in an HOL. Features of the HOL selected must include portability, parallelism, structuring capabilities, strong type-checking, and efficient object code.

GLOSSARY OF ABBREVIATIONS

AFM	Air Force Manual
AIMS	An Interactive Migration System
ANSI	American National Standards Institute
ARPA	Advanced Research Projects Agency
ARPANET	ARPA Network
ASPLE	A Simple Programming Language Example
BNF	Backus-Naur Form
C.mmp	Carnegie-Mellon Multiprocessor
CMOS	Complementary Metal Oxide Semiconductor
CODASYL	Committee on Data Systems Languages
CPU	Central Processing Unit
CRT	Cathode Ray Tube
CWS	Compiler Writing System
DAIS	Digital Avionics Information System
DBA	Data Base Administrator
DBMS	Data Base Management System
DBTG	Data Base Task Group
DDL	Data Definition Language
DEC	Digital Equipment Corporation
DMCL	Device Media Control Language
DML	Data Manipulation Language
ECMA	European Computer Manufacturers' Association
ETH	Eidgenossische Techniscle Hochschule (Federal Institute of Technology), Zurich, Switzerland
FEP	Front-End Processor
HDLC	High-level Data Link Control

HOL	High-Order Language
IC	Integrated Circuit
IDMS	Integrated Data Management System
IFIP	International Federation for Information Processing
IMP	Interface Message Processor
I/O	Input/Output
IWG6	IFIPS Working Group 6
JAVS	Jovial Automatic Verification System
JOCIT	Jovial Compiler Implementation Tool
LALR	Lookahead LR grammar
LL(K)	grammar whose strings are translatable from left to right using left reductions with k-symbol lookahead
LR(K)	grammar whose strings are translatable from left to right using right reductions with k-symbol lookahead
LSI	Large Scale Integration
MIC	Machine Independent Computer
MIL STD	Military Standard
MOL	Machine-Oriented Language
NCC	Norwegian Computing Center
NMOS	N-Channel Metal Oxide Semiconductor
OS	Operating System
OSI	Operating System Interface
PET	Program Evaluator and Tester
SPPTTG	Stored Data Definition and Translation Task Group
SDLC	Synchronous Data Link Control
SNA	System Network Architecture
SSG	Simula Standards Group

UDL	Universal Definition Language
VDL	Vienna Definition Language
XDMS	Experimental Data Management System

REFERENCES

[AED 1969]

"AED-0 Programmers Guide", Softech, Inc., Waltham, MA, December 1969.

[AFM 1967]

"Standard Computer Programming Language for Air Force Command and Control Systems", Air Force Manual No. 100-24, 1967.

[Aho and Ullman, 1973]

Aho, Alfred V. and Ullman, Jeffrey D. The Theory of Parsing, Translation, and Compiling. Vol. II: Compiling. Prentice-Hall, Inc., 1973.

[Aiello et. al., 1974]

Aiello, Luiga, et. al.,: "The Semantics of PASCAL in LCF", STAN-CS-74-447, Computer Science Department, Stanford University. (Also Stanford AI Lab. Memo 221 and ARPA Order No. 2494), August 1974.

[Aiello and Weyhrauch, 1974]

Aiello, L. and Weyhrauch, R.: "LCFsmall: An Implementation of LCF", STAN-CS-74-446, Computer Science Department, Stanford University, (Also Stanford AI Lab Memo 241 and ARPA Order Number 2495).

[Allen and Cocke 1976]

Allen, F.E. and Cocke, J., "A Program Data Flow Analysis Procedure", Communications of the ACM 19,3 (March 1976), 137-147

[Anderson 1973]

Anderson, Christine M., "Aerospace Higher Order Lanauage Processing", Technical Report AFAL-TR-73-151 (June 1973).

[Anderson et. al. 1974]

Anderson, E., Belz, F., and Blum, E. K.: "SEMANOL 73, A Metalanguage for Programming the Semantics of Programming Lanagues", TRW Technology Planning and Research Report TPR-TR-7, TRW Systems Group, June, 1974.

[Anderson et. al. 1976]

Anderson, E., Belz, F., and Blum, E. K.: "SEMANOL (73), A Metalanguage for Programming the Semantics of Programming Languages." Acta Informatica 6, FASC. 2, 109-131.

[ANSI X3.23-1974]

American National Standard Programming Language COBOL American National Standards Institute, 1974.

[ANSI/X3/SPARC 1975]

ANSI/X3/SPARC/STUDY Group. Data base systems, interim report. ACM/SIGMOD Newsletter (Dec. 1975).

[Astrahan et. al., 1976]

Astrahan, M. M. et al, "System R: Relational Approach to Database Management", ACM TODS, Vol, 1 #2, June 1976.

[Bachman 1975]

Bachman, C. W., "Trends in Database Management - 1975", AFIPS Vol. 44 NCC 1975.

[Backus and Heising 1964]

Backus, J.W. and Heising, W. P.: "FORTRAN" IEEE Transaction EC-13, pp. 382-385, 1964.

[Baker 1972a]

Baker, F.T., Chief programmer team management of production programming. IBM Syst. J., No. 1, 1976.

[Baker 1972b]

Baker, F.T. System quality through structured programming. Fall Joing Computer Conference, December 1972, 339-343.

[Barry and Naughton 1975]

Barry, Barbara S., and Naughton, John J., "Chief Programmer Team Operations Description", Structured Programming Series, Vol. 10, RADC-TR-74-300, January 1975.

[Beech and Marcotty 1973]

Beech, David and Marcotty, Michael. Unfurling the PL/I standard. SIGPLAN Notices 8, 10, October 1973.

[Bekic et. al, 1974]

Bekic, H., et al: "A Formal Definition of a PL/I Subset", IBM Technical Report 25.139, Vienna Laboratory, 1974.

[Berg 1975]

Berg, J. L. ed., "Data Base Directions: The Next Steps", NBS special publication 451, LOC 76-608219, Draft of a Report on the NBS/ACM Workshop in Fort Lauderdale, Fla., October 29-31, 1975.

[Berning 1973]

Berning, P. T. "A Standard for Language Implementation", RADC-TR-73-143, June 1973.

[Berning 1975a]

Berning, Paul T.: "A SEMANOL (73) Implementation Standard for JOVIAL (J73):, RADC-TR-75-211, Vol. I, 30 June 1975.

[Berning 1975b]

Berning, Paul T.: "SEMANOL (73) Reference Manual", RADC-TR-75-211, Vol. III, 30 June 1975.

[Berning 1975c]

Berning, Paul T.: "SEMANOL (73) Specification of JOVIAL (J73)", RADC-TR-75-211, Vol. IV, 30 June 1975.

[Berning 1975d]

Berning, Paul T.: "SEMANOL (73) Interpreter Documentation", RADC-TR-75-211, Vol. IV, 30 June 1975.

[Berning 1976]

Berning, Paul T.: TRW, Personal Communication, July 1976.

[Berry 1971]

Berry, Daniel: "Definition of the Contour Model in the Vienna Definition Language", TR-71-40, Center for Computer and Information Sciences, Brown University, April 1971.

[Berry 1974]

Berry, Daniel: "On the Design and Specification of the Programming Language OREGANO", UCLA-ENG-7388, Computer Science Department, UCLA, January 1974.

[Blum 1969a]

Blum, E. K.: "Towards a Theory of Semantics and Compilers for Programming Languages." Journal of Computer and System Sciences, Vol. 3, August 1969.

[Blum 1969b]

Blum, E. K.: "The Semantics of Programming Languages, Part I", TRW Software Series Report TRW-SS-69-01, TRW Systems Group, December 1969.

[Blum 1970]

Blum, E. K.: "The Semantics of Programming Languages, Part II", TRW Software Series Report TRW-SS-70-02, TRW Systems Group, December 1970.

[Blum 1973]

Blum, E. K.: "SEMANOL: A Formal System for the Semantics of Programming Languages", TRW Software Series Report TRW-SS-73-05, TRW Systems Group, August, 1973.

[Blum 1976]

Blum, E. K.: TRW, Personal Communication, July 1976.

[Bochmann 1974]

Bochmann, Gregor: "Attiibute Grammars and Compilation: Program Evaluation in Several Phases", Tech. Report #54, Departement D'Informatique, Universite De Montreal, 1974.

[Bochmann 1976a]

Bochmann, Gregor: "Semantic Evaluation From Left to Right", Communications of the ACM, Vol. 19, No. 2, February 1976, pp. 55-62.

[Bochmann, 1976b]

Bochmann, Gregor: University of Montreal, Personal Communication, July 1976

[Boehm 1973]

Boehm, Barry W.: "Software and Its Impact: A Quantitative Assessment", Datamation, May 1973, pp. 48-49.

[Boettcher 1974]

Boettcher, C.B.: "Program Evaluator and Tester", PET CDC Users Manual, McDonnell Douglas, Automation Company, Document No. M2085074, 1974.

[Bostrum 1971]

Bostrum, F. D.: "Higher Order Language Study for Avionics Programming", AFAL-TR-71-154, AD-884 708L, IBM Federal Systems Divison, June June 1971.

[Boyce and Chamberlin 1973]

Boyce, R.F. and Chamberlin, D.D.: "Using a Structured English Query Language as a Data Definition Facility", IBM Research Report RJ1318; (December 1973).

[Boyer et. al. 1975]

Boyer, R. S., Elspas, B., and Levitt, K.N.: "SELECT - A Formal System for Testing and Debugging by Symbolic Execution", Proceedings of the 1975 International Conference on Reliable Software, ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 234-245.

[Brinch Hansen 1975a]

Brinch Hansen, P.: "Concurrent PASCAL Report", Information Science, California Institute of Technology 1975.

[Brinch Hansen 1975b]

Brinch Hansen, :.: "The Solo Operating System: A Concurrent PASCAL Program", Information Science, California Institute of Technology, 1975.

[Brinch Hansen 1975c]

Brinch Hansen, P.: "The Solo Operating System: Job Interface", Information Science, California Institute of Technology, 1975.

[Brooker and Morris 1962]

Brooker, R. and Morris, D.: "A General Translation Program for Phrase Structure Languages", Journal of the ACM Vol. 9, No. 1, January 1972, pp. 1-10.

[Brooks et. al. 1976]

Brooks, N.B., Miller, E.F.Jr, and Wisehart, W.R., "JOVIAL Automated Verification System (JAVS)" RADC-TR-76-20, February 1976.

[Brosgol 1974]

Brosgol, B.M.: "The Role of the Compiler in Language Development", Intermetrics, Inc., November 1974.

[Brosgol et. al, 1975]

Brosgol, B.M., Dreisbach, T.A., Felty, J.L., Lexier, J.R., Palter, G.M., Grimes, D.E., Nestor, J.R.: "CS-4 Language Reference Manual and Operating System Interface", Intermetrics, Inc., Report IR-130-2, 1975.

[BSR X3.53 1975]

BSR X3.53: "Draft Proposed Standard Programming Language PL/I", BASIS/1-12, Computer and Business Equipment Manufacturers Association, February 1975.

[BSR X3.9 1976]

BSR X3.9: "Draft Prpposed ANS FORTRAN", BSR X3.9, X3J3/76, in SIGPLAN Notices, Vol. 11, No. 3, March 1976.

[Buchsbaum 1976]

Buchsbaum, Solomon, J.: "Lightwave Communications: An Overview", Physics Today, Vol. 29, No. 5, May 1976, pp. 23-25.

[Canaday et. al., 1974]

Canaday, R.H. et al: "A Back-End Computer for Data Base Management, CACM, Vol. 17, No. 10, October 1974.

[Chamberlin 1976]

Chamberlin, D.D., "Relational Data-Base Management Systems," ACM Computing Surveys, 8,1 (March 1976), 43-66.

[Chamberlin et. al. 1974]

Chamberlin et. al.: "A Deadlock-Free Scheme for Resource Locking in a Data-Base Environment", Information Processing 74, Proceedings of IFIP Congress, Stockholm, Sweden, August 5-10, 1974.

[Chirica et. al. 1973]

Chirica, Dreisbach, Martin, Peetz, and Sorkin: "Two Parallel EULER Run Time Models: The Dangling Reference, Imposter Environment, Label Problems", ACM-IEEE Symposium on High-Level Language Computer Architecture, College Park, Maryland, November 1973, pp. 141-151.

[Chirica and Martin 1975]

Chirica, L.M. and Martin D.F.: "An Approach to Compiler Correctness," Proceedings of International Conference on Reliable Software, SIGPLAN Notices, Vol. 10, Number 6, June 1975. pp. 96-103.

[Church 1941]

Church, A.: "The Calculi of Lambda Conversion", Princeton University Press, 1941.

[Chynoweth 1976]

Chynoweth, Alan G.: "The Fiber Lighguide", Physics Today, Vol. 29, No. 5., May 1976, pp. 28-37.

[Clark et. al. 1975]

Clark, D.D., Dennis, J.B., Hammer, M.M., Liskov, B.H., and Schroeder, M.D.: "Appraisal of the Woodenman and Recommended Plan for DoD High Order Language Development, Research and Consulting, Inc., 1975.

[Cleary 1969]

Cleary, J.G. "Process Handling on Burroughs B6500," in the Proceedings of the Fourth Australian Computer Conference, 1969.

[CODASYL 1971]

CODASYL: Data Base Task Group, April 71 Report, ACM, 1971.

[CODASYL 1973]

CODASYL Database Language Task Group, CODASYL COBOL database facility proposal, 1973.

[CODASYL 1976]

CODASYL COBOL Journal of Development, 1976.

[Codd 1970]

Codd, E.F.: "A Relational Model of Data for Large Shared Data Banks", CACM Vol. 13, No. 6, June 1970.

[Codd 1971a]

Codd, E.F.: "Further Normalization of the Data Base Relational Model", IBM Research Report RJ909, August 31, 1971

[Codd 1971b]

Codd, E.F.: "Normalized Data Based Structure: A Brief Tutorial", IBM Research Laboratory Proc. 1971 ACM SIGFIDET Workshop on Data Description Access and Control, San Diego, November 1971.

[Conwell 1976]

Conwell, Esther M.: "Integrated Optics", Physics Today, Vol. 29, No. 5, May 1976, pp. 48-59.

[Conway and Gries 1973]

Conway, Richard and Gries, David: An Introduction to Programming, Winthrop Publishers, Cambridge, Mass., 1973.

[Corbato 1969]

Corbato, F.J. PL/I as a tool for systems programming. Datamation, May 1969.

[Coral 1970]

Official Definition of Coral 66. Her Majesty's Stationery Office, 1970.

[Cray 1975]

"An Introduction to the CRAY-1 Computer", Cray Research, Inc., 1975.

[Dahl et. al. 1970]

Dahl, O. J., Myhrhaug, B. and Nygaard, K.: "Common Base Language", Norwegian Computing Center, 1970.

[Date and Codd 1974]

Date, C. J. and Codd, E.F.: "The Relational and Network Approaches: Comparison of the Application Programming Interfaces", IBM Research Report RJ1401 (#21706), June 1974.

[David 1968]

David, E.E. Comment in [NATO 1968], 56-57.

[DEC 1973]

PDP11/05/10/35/40 Processor Handbook. Digital Equipment Corp. 1973.

[DeMillo 1975]

DeMillo, R.A.: "Primitives for Tactical Real Time Control Languages Based on SIMULA 67", U S Army Electronics Command, Center for Tactical Computer Sciences, Report No. 50, 1975.

[Denning 1970]

Denning, P.J.: "Virtual Memory", Computer Surveys, Vol. 2, September, 1970, pp. 153-189.

[Dennis 1972]

Dennis J.B.: "Modularity", Software Engineering, An Advanced Course, Springer-Verlag 1972.

[DeRemer 1969]

DeRemer, F.L., Practical Translators for LR(k) Languages, PhD. Thesis M.I.T., Cambridge, Mass., October 1969.

[DeRemer 1971]

DeRemer, F. L., "Simple LR(k) Grammars", Communications of the ACM 14, pp. 453-460. July 1971.

[DeRemer and Kron 1975]

DeRemer, F., and Kron, H.: "Programming-in-the-Large versus Programming-in-the-Small", Proceedings of the 1975 International Conference on Reliable Software, ACM SIGPLAN Notices, Vol 10, No. 6, June 1975, pp. 114-121.

[DeTreville and Tersoff 1976]

DeTreville, J.D., Tersoff, M.: "Major Implementation Decisions", M.I.T.'s Sloan School of Management, Report F.O.S. No. R02, Rev. D., 1976.

[Dijkstra 1968a]

Dijkstra, E. W.: "Cooperating Sequential Processes", in Genuys, F. (ed.), Programming Languages, Academic Press, 1968.

[Dijkstra, 1968b]

Dijkstra, E.W.: "The Structure of 'THE' Multiprogramming System", CACM 11, 5, 1968.

[Dijkstra 1971]

Dijkstra, E.W.: "Hierarchical Ordering of Sequential Processes", Acta Informatica, Vol. I, No. 2, 1971.

[Dijkstra 1972a]

Dijkstra, E.W.: "Notes on Structured Programming", in Dahl, J.J., Dijkstra, E.E., and Hoare, C.A.R.: Structured Programming, Academic Press, 1972, pp. 1-82.

[Dijkstra 1972b]

Dijkstra, E.W.: "The Humble Programmer", Communications of the ACM. Vol. 15, No. 10, October, 1972, pp. 859-866.

[Dijkstra 1976]

Dijkstra E.W.: "A Discipline of Programming", Prentice Hall, Inc., 1976.

[Donahue 1975a]

Donahue, James: "A Child's Guide to Scottery", Tech. Note 1, Computer Systems Research Group, University of Toronto, August 1975.

[Donahue 1975b]

Donahue, James: "Complementary Definitions of Programming Language Semantics", Tech. Report CSRG-62, Computer Systems Research Group, University of Toronto, November 1975.

[Donovan and Ledgard 1967]

Donovan J., and Ledgard, J.: "A Formal System for the Specification of the Syntax and Translation of Computer Languages", 1967 FJCC Proceedings, AFIPS Press, Vol. 31, 1967.

[Douque and Nijssen 1975]

Douque, B.C.M. and Nijssen, G.M.: Data Base Description. Proceedings of the IFIP TC-2 Special Working Conference on Data Base Description Elsevier-North Holland Publishing Co., 1975.

[Dreisbach 1972]

Dreisbach, Timothy A.: "A Declarative Semantic Definition of PL360", UCLA-ENG-7289, Computer Science Department, UCLA, October 1972.

[Dunbar 1975]

Dunbar, Terry L.: "JOCIT JOVIAL Compiler Implementation Tools," RADC-TR-74-322, January 1975.

[Economist 1976]

"Microeconomics", The Economist, August 1976, pp. 52-53

[Engles 1971]

Engles, R.W.: "An Analysis of the April 1971 DBTG Report", Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, November 1971.

[Enslow 1975]

Enslow, P.H.: "CORAL 66 -- The U.K. Standard Programming Language for Weapons Systems", in Implementation Languages for Real-Time Systems, Part I. Standardization -- Its Implementation and Acceptance, European Research Office, U.S. Army Research and Development Group (Europe), Tech. Report No. ERO-2-75, Vol. I, 1975.

[Evans and Morgan 1976]

Evans, A., Jr., and Morgan C.R.: "Development of a Communications Oriented Language, Part II: Development of the COL Concept and Syntax", Bolt Beranek and Newman, Inc., Report No. 3261, March 1976.

[Fang 1972]

Fang, I.: "FOLDS A Declarative Formal Language Definition System", STAN-CS-72-329, Computer Science Department, Stanford University, 1972.

[Feldman 1966]

Feldman, J. A.: "A Formal Semantics for Computer Languages and Its Application in a Compiler-Compiler", Communications of the ACM, Vol. 9, No. 1, January 1966, pp. 3-9.

[Feustel 1973]

Feustel, E. A., "On the Advantages of Tagged Architecture", IEEE Transactions on Computers, Vol. C-22, No. 7, July, 1973.

[Fisher 1976]

Fisher, David A.: A Common Programming Language for the Department of Defense -- Background and Technical Requirements, Institute for Defense Analyses, Paper P-1191, June, 1976.

[Floyd 1967]

Floyd, R. W.: "Assigning Meanings to Programs", in Proceedings of the Symposium on Applied Mathematics, Vol. 19, Amer. Math. Soc., 1967.

[Frampton 1976]

Frampton, L. D., Chairperson of X3J1 Committee, in personal telephone conversation, August 3, 1976.

[Fry and Sibley 1976]

Fry, J. P., and Sibley, E. H.: "Evolution of Data-Base Management Systems", ACM Computing Surveys, Vol. 8, No. 1, March 1976.

[Fry et. al. 1972]

Fry, J.P., Smith, D.P., and Taylor, R.W.: An approach to stored data definition and translation. ACM-SIGFIDET Workshop on Data Description, Access, and Control (November -December 1972), 13-56.

[Gannon and Horning 1975]

Gannon, J. D., and Horning, J. J.: "The Impact of Language Design on the Production of Reliable Software", Proceedings of the 1975 International Conference on Reliable Software, ACM SIGPLAN Notices, Vol 10, No. 6, June 1975, pp. 10-22.

[Garwick 1966]

Garwick, J. V.: "The Definition of Programming Languages by Their Compilers", in [Steel 1966], pp. 139-147.

[Gazette 1976]

DEC System-10 SIMULA Gazette, Vol. 2, No. 4, April 1976, pp. 7-8.

[Gerhart 1971]

Gerhart, S.: "Definition of APL in the Vienna Language", Carnegie-Mellon University, 1971.

[Gerritsen 1975]

Gerritsen, R.: A preliminary system for the design of DBTG data structures. Comm. ACM (October 1975).

[Goodenough 1975]

Goodenough, J.B.: "Exception Handling: Issues and a Proposed Notation," CACM 18, 12, 1975.

[Graham 1968]

Graham, R. Comment in [NATO 1968], 57.

[Graham and Rhodes 1975]

Graham, Susan L., and Rhodes, Steven P.: "Practical Syntactic Error Recovery", Communications of the ACM 18, 11, November 1975, pp. 639-650.

[Graham et. al. 1976]

Graham, S. L., Harrison, M. A., Ruzzo, W. L., "On-line Context-Free Recognition in Less Than Cubic Time", 8th Annual ACM Symposium on Theory of Computing, Hershey, Pa. May 1976.

[Graham and Wegman 1976]

Graham, Susan L., and Wegman, Mark: "A Fast and Usually Linear Algorithm for Global Flow Analysis", Journal of the ACM Vol. 23, No. 1, January 1976, pp. 172-202.

[Griffiths 1974]

Griffiths, M.: "LL(1) Grammars and Analysers", Compiler Construction: An Advanced Course, Lecture Notes in Computer Science, Vol. 21, Springer-Verlag, Berlin-Heidelberg-New York, 1974, pp. 57-84.

[Hewitt et. al. 1973]

Hewitt, Carl, Bishop, Peter, Greiff, Irene, Smith, Brian, Matson, Todd, and Steiger, Richard: "Actor Induction and Meta-evaluation", in Conference Record of ACM Symposium on Principles of Programming Languages, October 1-3, 1973, pp. 153-168.

[Hoare 1969a]

Hoare, C. A. R.: "An Axiomatic Basis for Computer Programming", Communications of the ACM, Vol. 12, No. 10, October 1969, pp. 576-583.

[Hoare 1969b]

Hoare, C. A. R.: "An Axiomatization of the Data Definition Features of PASCAL", NATO Science Committee, Techniques in Software Engineering working material, Vol 2, 1969.

[Hoare 1971a]

Hoare, C. A. R.: "Procedures and Parameters: An Axiomatic Approach", in Engeler (ed.), "Semantics of Algorithmic Languages", Springer Notes in Mathematics, Vol. 188, 1971, pp. 102-116.

[Hoare 1971b]

Hoare, C. A. R.: "Proof of a Program: FIND", Communications of the ACM, Vol. 14, No. 1, January 1971, pp. 39-45.

[Hoare 1972a]

Hoare, C. A. R.: "A Note on the 'FOR' Statement", BIT, No. 12, 1972, pp. 334-341.

[Hoare 1972b]

Hoare, C. A. R.: "Proof of Correctness of Data Representations", Acta Informatica, Vol. I, Springer-Verlag, 1972, pp. 271-281.

[Hoare 1973]

Hoare, C. A. R.: "Critique of 'The JOVIAL (J73) Computer Programming Language Specification 1971 January 1'", Rome Air Development Center, New York.

[Hoare 1974a]

Hoare, C. A. R., "Monitors: An Operating System Structuring Concept", Communications of the ACM, 17, 10 (October 1974), 549-557.

[Hoare 1974b]

Hoare, C.A.R.; "Hints on Programming Language Design," in Computer Systems Reliability, Infotech State of the Art Report, No. 20, 1974.

[Hoare 1975]

Hoare, C.A.R.: "Hints on the Design of a Programming Language for Real-Time Command and Control," in Implementation Languages for Real-Time Systems, U.S. Army European Research Office Tech. Report No. ERO-2-75, Vol. 2, 1975.

[Hoare and Wirth 1973]

Hoare, C. A. R., and Wirth, N.: "An Axiomatic Definition of the Programming Language PASCAL", Acta Informatica, Springer-Verlag, 1973, pp. 335-355.

[Hoare and Lauer 1974]

Hoare, C. A. R., and Lauer, P. F.: "Consistent and Complementary Formal Theories of the Semantics of Programming Languages", Acta Informatica, Vol. 3, No. 2, Springer-Verlag, 1974, pp. 135-154.

[Holt 1973]

Holt, Richard C.: "Teaching the Fatal Disease (or) Introductory Programming Using PL/I", SIGPLAN Notices, Vol. 8, No. 5, May 1973, pp. 8-23.

[Horning 1974a]

Horning, J. J.: "LR Grammar and Analysers", Compiler Construction: An Advanced Course, Lecture Notes in Computer Science, Vol. 21, Springer-Verlag, 1974, pp. 85-108.

[Horning 1974b]

Horning, J. J.: "Structuring Compiler Development", Compiler Construction: An Advanced Course, Lecture Notes in Computer Science, Vol. 21, Springer-Verlag, 1974, pp. 498-513.

[Housel et. al. 1974]

Housel, B.C., Sum, V.Y., and Shu, N.: Architecture of an interactive migration system (ADMS). ACM SIGMOD Workshop on Data Description, Access and Control (May 1974), 157-170.

[IBM 1966]

IBM PL/I Definition Group: "Formal Definition of PL/I", IBM Tech. Report 25.071, IBM Vienna Laboratory, 1966.

[IBM 1974]

IBM System/370 Principles of Operation, Fourth Edition. IBM System-Products Division, September 1974.

[IBM 1975a]

IBM Corporation: "Systems Network Architecture: General Information", 1975.

[IBM 1975b]

IBM Corporation: "Synchronous Data Link Control: General Information", 1975.

[Ichbiah et al 1974]

Ichbiah, J.D., Rissen, J.P., Helliard, J.C., and Cousot, P.: "The System Implementation Language LIS" CII - Technical Report TR4549, EIEN, December 1974.

[INFOTECH 1974]

"INFOTECH State of the Art Report 17: Computer Design", Infotech Information Ltd., 1974.

[INFOTECH 1975]

"INFOTECH State of the Art Report 25: Data Base Systems", Infotech Information Ltd., 1975.

[Intermetrics 1975]

"On the Performance of the HAL/S-FC Compiler," Intermetrics, Inc., Report IR-162, 1975.

[Irons 1961]

Irons, E. T.: "A Syntax-Directed Compiler for ALGOL 60", Communications of the ACM, Vol. 4, No. 1, January 1961, pp. 51-55.

[Irons 1963]

Irons, E. T.: "Towards More Versatile Mechanical Translators", Proceedings of the Symposium on Applied Mathematics, Vol. 15, 1963, pp. 41-50.

[Iverson 1962]

Iverson, K. E., A Programming Language. John Wiley & Sons, New York, 1962.

[Ives 1976]

Ives, John: RADC, Personal Communication, July 1976

[Jelinski 1976]

Jelinski, Z.: McDonnell Douglas, Personal Communication, July 1976.

[Jensen and Wirth 1975]

Jensen, K., and Wirth, N.: "PASCAL User Manual and Report", Lecture Notes in Computer Science, Vol. 18, Springer-Verlag, 1975.

[Johnson 1975]

Johnson, H.R., A Schema Report Facility for a CODASYL Based Data Definition Language. IFIP Tc-2 Special Working Conference on Data Base Description, 299-328.

[Johnston 1970]

Johnston, John: "The Contour Model of Block Structured Processes", Report 70-C-366, General Electric Research and Development Center, Schenectady, New York, October, 1970.

[Kahn 1972]

Kahn, Robert E.: "Resource-Sharing Computer Communications Networks", Proceedings of the IEEE, Vol. 60, No. 11, November 1972, pp. 1397-1407.

[Kam and Ullman 1976]

Kam, John B. and Ullman, Jeffrey D., "Global Flow Analysis and Interactive Algorithms", Journal of ACM, Vol. 23, No. 1, January 1976.

[Kelly 1974]

Kelly, J. R.: "A Definition of ALGOL 68 in the Vienna Definition Language", Brown University, 1974.

[Kerschberg et. al. 1976]

Kerschberg, L., et. al.: "A Taxonomy of Data Models", University of Toronto Technical Report CSRG-70, May 1976.

[Kimbleton and Mandell 1976]

Kimbleton, Stephen R. and Mandell, Richard L.: "A Perspective on Network Operating Systems", National Computer Conference Proceedings 1976, pp. 551-559.

[Kimbleton and Schneider 1975]

Kimbleton, Stephen R. and Schneider, G. Michael: "Computer Communications Networks: Approaches, Objectives and Performance Considerations", ACM Computing Surveys, Vol. 7, No. 3, September 1975, pp. 129-173.

[King 1976]

King, James C.: "Symbolic Execution and Program Testing", Communications of the ACM Vol. 19, No. 6, July 1976, pp. 385-394.

[Knuth 1965]

Knuth, D. E., "On the Translation of Languages from Left to Right", Information and Control 8, pp. 607-639. 1965.

[Knuth 1968]

Knuth, Donald: "Semantics of Context-Free Languages", Math Systems Theory, Springer-Verlag, Vol. 2, No. 2, 1968, pp. 127-145.

[Knuth 1969]

Knuth, Donald E: "The Art of Computer Programming, Vol. 2: Seminumerical Algorithms", Addison Wesley 1969.

[Kosinski 1976]

Kosinski, M. S., SPL/I Language Reference Manual, Intermetrics, Inc. Report No. IR-172-1, July, 1976.

[Koster 1971]

Koster, C.: "Affix Grammars", in "ALGOL 68 Implementation", North Holland Publishing Company, Amsterdam, 1971.

[Kosy 1974]

Kosy, D. W.: "Air Force Command and Control Information Processing in the 1980s: Trends in Software Technology", Rand Corporation, R-1012-PR, June 1974.

[Kotok 1974]

Kotok, A.: "Future Trends in Computer Architecture", INFOTECH State of the Art Report 17: Computer Design, Infotech Information 1974, pp. 327-342.

[Kressel et. al. 1976]

Kressel, Henry, Ladany, Ivan, Ettenberg, Michael, and Lockwood, Harry: "Light Sources", Physics Today, Vol. 29, No. 5, May 1976, pp. 38-47.

[LaLonde 1971]

LaLonde, W. R., An Efficient LALR Parser Generator, Univ. of Toronto CSRG-2, 1971.

[Landin 1964]

Landin, Peter: "The Mechanical Evaluation of Expressions", Computer Journal, Vol. 6, No. 4, January 1964, pp. 308-320.

[Landin 1965]

Landin, Peter: "A Correspondence Between ALGOL 60 and Church's Lambda Notation", Communications of the ACM, Vol. 7, Nos. 2, 3 (Feb., Mar.) 1965, pp. 89-101, 158-165.

[Landin 1966]

Landin, Peter: "A Formal Description of ALGOL 60", in [Steel 1966], pp. 266-294.

[LaPadula and Loring 1976]

LaPadula, L. J., and Loring, P. L. "Air Force Programming Languages: Standards, Use, and Selection", the MITRE Corp., MTR-3169, January 1976.

[Lauer 1968]

Lauer, P.: "Formal Definition of ALGOL 60", IBM Tech. Report 25.088, IBM Vienna Laboratory, 1968.

[Lecarme and Bochmann 1974]

Lecarme, Oliver, and Bochmann, Gregor V.: "A (Truly) Usable and Portable Compiler Writing System", Information Processing 74, Proceedings of IFIP Congress, Stockholm, Sweden, August 5-10, 1974.

[Ledgard 1969]

Ledgard, Henry: "A Formal System for Defining the Syntax and Semantics of Programming Languages", MAC-TR-60, Project MAC, M. I. T., 1969.

[Ledgard 1972]

Ledgard, Henry: "Production Systems: A Formalism for Defining the Syntax and Translation of Practical Programming Languages", Tech. Report 24, Dept. of Electrical Engineering, Johns Hopkins University, December 1972.

[Ledgard 1974]

Ledgard, Henry: "Production Systems: Or Can We Do Better Than BNF?", Communications of the ACM, Vol. 17, No. 2, February 1974, pp. 94-102.

[Ledgard 1975]

Ledgard, Henry: "Production Systems: A Formal Notation for Defining the Syntax and Translation of Programming Languages", COINS Tech. Rep. 75A-4, University of Massachusetts, 1975.

[Ledgard 1976]

Ledgard, Henry: University of Massachusetts, Personal Communication, July 1976.

[Leinius 1970]

Leinius, R. P.: "Error Detection and Recovery for Syntax Directed Compiler Systems", Ph.D. Thesis, Computer Science Department, University of Wisconsin (Madison) 1970.

[Levy 1971]

Levy, J. P.: "Automatic Correction of Syntax Errors in Programming Languages", Ph.D. Thesis, Cornell University, Computer Science Department Technical Report TR71-116, December 1971.

[Lewis and Stearns 1968]

Lewis, P. M., and Stearns, R. E.: "Syntax-Directed Transduction", Journal of the ACM, Vol. 15, No. 3, July 1968, pp. 465-488.

[Lewis et. al. 1973]

Lewis, P., Rosenkrantz, D., and Stearns, R.: "Attributed Translations", ACM Symposium on the Theory of Computing, Austin, Texas, April 1973.

[Lexier 1975]

Lexier, J. R.: "Considerations in the Choice of PL/I for a Real-Time Command and Control Language", Intermetrics, Inc., Report No. IR-125-1, July 1975.

[Linden 1976]

Linden, Theodore A.: "The Use of Abstract Data Types to Simplify Program Modifications", Institute for Computer Sciences and Technology, National Bureau of Standards, 1976.

[Liskov 1972]

Liskov, B. H.: "The Design of the Venus Operating System", CACM, Vol. 15, No. 3, 1972.

[Liskov 1976]

Liskov, B. H.: "An Introduction to CLU", M. I. T. Laboratory for Computer Science, Computation Structures Group Memo 136, 1976.

[London, 1971]

London, R.L." Correctness of Two Compilers for a LISP Subset", A.I., Memo 151, Stanford 1971.

[Lucas et. al. 1968]

Lucas, P. et. al.: "Method and Notation for the Formal Definition of Programming Languages", IBM Tech. Report 25.087, IBM Vienna Laboratory, 1968.

[Lucas and Walk 1969]

Lucas, P, and Walk, K.: "On the Formal Description of PL/I", in Annual Review of Automatic Programming, Vol. 6, No. 3, Pergamon Press, New York, 1969.

[Lucking 1974]

Lucking, J.R., Data Base Languages in Particular DDL Development at CODASYL. ACM SIGMOD Workshop on Data Description, Access and Control (May 1974), 35-50.

[Luppino and Smith 1974]

Luppino, F. M., and Smith, R. L.: "Programming Support Library (PSL) Functional Requirements", Structured Programming Series, Vol. 5, RADC-TR-74-300, July 1974.

[McCarthy 1960]

McCarthy, J.: "Recursive Functions of Symbolic Expressions and Their Computation by Machine", Communications of the ACM, Vol. 3, No. 4, April 1960, pp. 184-195.

[McCarthy 1963a]

McCarthy, J.: "Towards a Mathematical Theory of Computation", 1962 IFIP Conference Proc., North Holland Pub. Co., Amsterdam, 1963.

[McCarthy 1963b]

McCarthy, J. A Basis for a Mathematical Science of Computation", in Braffort and Hirschberg (eds.): "Formal Programming Languages", North Holland Publishing Co., Amsterdam, 1963.

[McCarthy 1966]

McCarthy, J.: "A Formal Description of a Subset of ALGOL", in [Steel 1966], pp. 1-12.

[McDermott and Sussman 1974]

McDermott, Drew V. and Sussman, Gerald J: "The Conniver Reference Manual", M.I.T. A.I. Memo N.259a, May 1972, updated January 1974.

[McKeeman et. al. 1968]

McKeeman, W. M., Horning, J. J., Nelson, E. C., and Wortman, D. B.: "The XPL Compiler Generator System", Proceedings of the Fall Joint Computer Conference, 1968, pp. 617-635.

[McKeeman et. al. 1970]

McKeeman, W. M., Horning, J. J., Wortman, D. B.: "A Compiler Generator", Prentice-Hall, Inc., 1970.

[Macri 1976]

Macri, P. P.: "Deadlock Detection and Resolution in a CODASYL Based Data Management System", ACM SIGMOD International Conference on Modification of Data, June 1976.

[Manna 1969]

Manna, Z.: "Properties of Programs and the First-Order Predicate Calculus", Journal of the ACM, Vol. 16, No. 2, April 1969, pp. 244-255.

[Marcotty 1975]

Marcotty, Michael: "Suitability of PL/I as the Army Programming Language", CENTACS software report no. 47, U. S. Army Electronics Command, June 1975.

[Marcotty et. al. 1976]

Marcotty, M., Ledgard, H., and Bochmann, G.: "A Sampler of Formal Definitions", Preliminary Draft of Paper to Appear in Computing Surveys, Draft of February 1976.

[Mertan and Fry 1974]

Mertan, A.G. and Fry, J.P. A data description approach to file translation. ACM SIGMOD Workshop on Data Description, Access and Control, May 1974.

[Michaels et. al. 1976]

Michaels, A. S. et. al.: "A Comparison of Relational and CODASYL Approaches to Data-Base Management", ACM Computing Surveys, Vol. 8, No. 1, March 1976.

[Miller et. al. 1973]

Miller, J. S., Mikkelsen, C., Nestor, J. R., Brosgol, B. M., Pepe, J. T., and Fourer, R.: "CS-4 Language Reference Manual", Intermetrics, Inc., December 1973.

[Mills 1971]

Mills, H. D.: "Chief Programmer Teams: Principles and Procedures", Report No. FSC 71-5108, IBM FSD, Gaithersburg, Maryland, June 1971.

[Mills 1972]

Mills, D. L.: "Communication Software", Proceedings of the IEEE, Vol. 60, No. 11, November 1972, pp. 1333-1341.

[Mills 1975]

Mills, Harlan D., How to write correct programs and know it. International Conference on Reliable Software, April 1975.

[Mills 1976]

Mills, D. L.: "An Overview of the Distributed Computer Network", National Computer Conference Proceedings 1976, pp. 523-531.

[Milner 1972a]

Milner, R.: "Logic For Computable Functions Description of a Machine Implementation", Stanford University, AI Lab. Memo 196, 1972.

[Milner 1972b]

Milner, R.: "Implementation and Applications of Scott's Logic for Computable Functions", Proceedings of the Las Cruces Conference on Proving Assertions about Programs, New Mexico, 1972, pp. 1-5.

[Morel and Renvoise 1976]

"A Global algorithm for the elimination of partial redundancies", 2nd International Symposium on Programming, Paris, April 1976.

[Nakata 1967]

Nakata, I.: "A note on compiling algorithms for arithmetic expressions," Communications of ACM, Vol 10, No. 8, August 1967, pp. 492-494.

[NATO 1968]

Software Engineering. NATO Science Committee, October 1968, 56-57.

[Neuhold 1971]

Neuhold, E. J.: "The Formal Description of Programming Languages", IBM Systems Journal, Vol. 10, No. 2, 1971.

[Nijssen 1974]

Nijssen, C.M. Data structuring in the DDL and relational data model. Proceedings of the IFIP Working Conference on Data Base Management, April 1974, pp. 363-384.

[Noyes 1976]

Noyes, J.: Presentation to the U. S. DoD representatives on CORAL 66, in International Efforts Pertaining to Standardization of Real-Time High-Order Languages -- Selected Documentation, B. Zempolich and R. Kahane (eds.), February 1976.

[Olle 1974]

Olle, T.W., Current and future trends in data base management systems. Proceedings of IFIP Congress 1974; Information Processing 1974.

[Ozkarahan et. al. 1976]

Ozkarahan, E. A., et. al.: "Performance Evaluation of a Relational Associative Processor", University of Toronto Technical Report CSRG-65, February 1976.

[Palme 1975]

Palme, J.: "Language for Reliable Software", in Implementation Languages for Real-Time Systems, Part II, Language Design -- General Comments, European Research Office, U. S. Army Research and Development Group (Europe), Tech. Report No. ERO-2-75-Vol. 2, 1975.

[Parmelee et. al. 1972]

Parmelee, R. P., Peterson, T. I., Tilman, C. C., and Hatfield, D. J.: "Virtual Storage and Virtual Machine Concepts", IBM Systems Journal, Vol. 11, No. 2, 1972, pp. 99-130.

[Parnas 1971]

Parnas, D. L.: "Information Distribution Aspects of Design Methodology", Computer Science Department, Carnegie-Mellon University, February 1971.

[Parnas 1972]

Parnas, D. L.: "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, December 1972.

[Parnas 1976]

Parnas, D. L.: "On the Design and Development of Program Families", IEEE Transactions of Software Engineering, Vol. 2, No. 1, 1976.

[Peterson 1972]

Peterson, T. G.: "Syntax Error Detection, Corrections and Recovery in Parsers", Ph.D. Thesis, Stevens Institute of Technology, Hoboken, New Jersey, 1972.

[Plagman and Altshuler 1972]

Plagman, B.K. and Altshuler, G.P.: A data dictionary/directory system within the context of an integrated corporate data base. Fall Joint Computer Conference, 1972, pp. 1133-1140.

[Poole 1974]

Poole, P. C.: "Portable and Adaptable Compilers", Compiler Construction: An Advanced Course, Lecture Notes in Computer Science, Vol. 21, Springer-Verlag, 1974.

[Post 1943]

Post, E. L.: "Formal Reductions of the General Combinatorial Decision Problem", Amer. J. of Math., Vol. 65, 1943, pp. 197-215.

[Pouzin 1975]

Pouzin, Louis: "Standards in Data Communications and Computer Networks", in Fourth Data Communications Symposium Proceedings, October 1975, pp. 2-8 to 2-12.

[RADC 1976]

"Jovial J73/I Specification", Rome Air Development Center, 1976.

[Radin and Rogoway 1965]

Radin, G. and Rogoway, H.P. NPL: highlights of a new programming language. Comm. ACM, 8, 1, January 1965.

[Radzeijowsky 1969]

Radzeijowsky, R.R.; "On arithmetic expressions and trees," Communications of ACM, Vol. 12, No. 2, February 1969, pp. 80-84.

[Ralston 1973]

Ralston, A.: "The Future of Higher Level Languages (in Teaching)", Proceedings of the International Computing Symposium, 1973, pp. 1-10.

[Reifer 1975]

Reifer, D. J.: "Automated Aids for Reliable Software", Proceedings of the 1975 International Conference on Reliable Software, ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 131-142.

[Reifer 1976]

Reifer, D. J.: "The Structured FORTRAN Dilemma", SIGPLAN Notices, Vol. 11, No. 2, February 1976, pp. 30-32.

[Robinson 1976]

Robinson, J. Allen: Syracuse University, Personal Communication, July 1976.

[Rosenkrantz and Stearns 1969]

Rosenkrantz, D. J., and Stearns, R. E., "Properties of Deterministic Top Down Grammars", ACM Symposium on Theory of Computing, Marina del Rey, Calif. May 1969.

[Rosenthal 1976]

Rosenthal, Robert: "Network Access Techniques -- A Review", National Computer Conference Proceedings 1976, pp. 495-500.

[Rubey 1968]

Rubey, Raymond J.: A comparative evaluation of PL/I. Datamation, December 1968.

[Rybczynski et. al. 1976]

Rybczynski, A., Wessler, B., Despres, R., and Wedlake, J.: "A New Communication Protocol for Accessing Data Networks -- The International Packet-Mode Interface", Proceedings of the National Joint Computer Conference, 1976, pp. 477-482.

[Sammett 1969]

Sammett, Jean E. Programming Languages: History and Fundamentals. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1969.

[Schneck 1975]

Schneck, P. B.: "Movement of Implicit Parallel and Vector Expressions Out of Program Loops", SIGPLAN Notices, Vol. 10, No. 3, March 1975, pp. 103-106.

[Schubert 1974]

Schubert, R. F.: "Directions in Data Base Management Technology", Datamation, September 1974.

[Scott 1970]

Scott, Dana: "Outline of a Mathematical Theory of Computation", Programming Research Group Monograph PRG-2, Oxford University Computing Laboratory, November 1970.

[Scott and Strachey 1971]

Scott, Dana, and Strachey, Christopher: "Towards a Mathematical Semantics for Programming Languages", Programming Research Group Monograph PRG-6, Oxford University Computing Laboratory, August 1971.

[Senko 1975]

Senko, M.E. Data description language in the concept of a multi level structured description: DIAM II with FORAL, In Data Base Description, June 1975, Douque and Jujssen, eds., 239-258.

[Serlin 1972]

Serlin, O.: "Scheduling of Time Critical Processes", Proceedings of the Spring Joint Computer Conference, 1972, pp. 925-932.

[Sethi and Ullman 1970]

Sethi, R. and Ullman, J.D.: "The generation of Optimal Code for Arithmetic Expression", Journal of ACM. Vol. 17, No. 4, pp. 715-728.

[Sevcik et. al. 1972]

Sevcik, K. C., Atwood, J. W., Grushcow, M. S., Holt, R. C., Horning, J. J., and Tsichritzis, D.: "Project SUE as a Learning Experience", AFIPS Vol. 41, FJCC, 1972.

[Shaw 1968]

Shaw, Christopher J. PL/I for C & C? Datamation, December 1968.

[Shooman and Bolsky 1975]

Shooman, M. L., and Bolsky, M. I.: "Types, Distribution, and Test and Correction Times for Programming Errors", Proceedings of the 1975 International Conference on Reliable Software, ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 347-357.

[Shorter 1976]

Shorter, D.: Presentation to the U. S. DoD representatives on CORAL 66, in International Efforts Pertaining to Standardization of Real-Time High-Order Languages -- Selected Documentation, B. Zempolich and R. Kahane (eds.), 27 February 1976.

[Sibley and Taylor 1970]

Sibley, E.H. and Taylor, R.W.: "Preliminary Discussion of a General Data-to-Storage Structure Mapping Language", ACM SIGFIDET Workshop on Data Description and Access, 1970.

[Sibley and Taylor 1973]

Sibley, E.H. and Taylor, R.W.: A data definition and mapping language Comm. ACM 16, 12, December 1973, pp. 750-759.

[SIGFIDET 1970]

Proceedings of 1970 ACM SIGFIDET Workshop on Data Description and Access, Texas, November 1970.

[SIGFIDET 1972]

Proceedings of 1972 ACM SIGFIDET Workshop on Data Description and Access, Denver, November 1972.

[SIGPLAN 1974]

Notice in SIGPLAN Notices, Vol. 9, No. 3, March 1974, p. 3.

[Smith 1971]

Smith, D.P., Ph.D. Th., University of Pennsylvania, 1971.

[Smith 1974]

Smith, D.C.P. From a data description point of view. ACM SIGMOD Workshop on Data Description, Access and Control, May 1974.

[Smith 1975]

Smith, Ronald L.: "Validation and Verification Study", Structured Programming Series, Vol. 15, RADC-TR-74-300, May 1975.

[Smith and Chang 1975]

Smith, J.M. and Chang, P.Y.T., Optimizing the performance of a relational algebra database interface. Comm. ACM 18, 10, October 1975, pp. 568-579.

[Smullyan 1961]

Smullyan, R. M.: "Theory of Formal Systems", Annals of Mathematical Studies, No. 47, Princeton University Press, 1961.

[Softech 1976]

"JOVIAL/J3B Language Specification Extension 2", Report 2044-4.3, Softech, Inc., 1976.

[Spoonley 1974]

Spoonley, N.: "Design Requirements for Network Operation", INFOTECH State of the Art Report 17: Computer Design, Infotech Information, 1974, pp. 399-411.

[Stacey 1974]

Stacey, G. M.: "A FORTRAN Interface to The CODASYL DBTG Specifications", The Computer Journal, Vol. 17, No. 2, May 1974.

[Standish 1975]

Standish, T. A.: "Extensibility in Programming Language Design", SIGPLAN Notices, Vol. 10, No. 7, July 1975, pp. 18-21.

[Steel 1966]

Steel, T. B. Jr., (ed.): "Formal Language Description Languages for Computer Programming", Proceedings of the 1964 IFIP Working Conference on Formal Language Description Languages, North Holland Pub. Co., Amsterdam, 1966.

[Steel 1967]

Steel, T. B., Jr.: "Standards for Computers and Information Processing", in Alt and Rubinoff (eds.), "Advances in Computers", Vol. 8, Academic Press, New York, 1967.

[Steele and Sussman 1976]

Steele, Guy L. and Sussman, Gerald J.: "Lambda, The Ultimate Imperative", M.I.T. A.I. Memo No. 353, March 1976.

[Strachey 1966]

Strachey, Christopher: "Towards a Formal Semantics", in [Steel 1966], pp. 198-220.

[Strachey 1973]

Strachey, Christopher: "The Varieties of Programming Language", Programming Research Group Monograph PRG-10, Oxford University Computing Laboratory, March 1973.

[Strachey and Wadsworth 1973]

Strachey, C., and Wadsworth, C.: "Continuations: A Mathematical Semantics for Handling Full Jumps", Programming Research Group Monograph PRG-13, Oxford University Computing Laboratory, 1973.

[Stucki and Foshee 1975]

Stucki, L. G., and Foshee, G. L.: "New Assertion Concepts for Self-Metric Software Validation", Proceedings of the 1975 International Conference on Reliable Software, ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975, pp. 59-71.

[Sussman and Steele 1975]

Sussman, Gerald J. and Steele, Guy L.: "Scheme, An Interpreter for Extended Lambda Calculas," M.I.T. A.I. Memo No. 349, December 1975.

[Tate 1974]

Tate, D.: "Pipelining: Design Problems and Implications", INFOTECH State of the Art Report 17: Computer Design, Infotech Information, 1974, pp. 251-280.

[Taylor 1971]

Taylor R.W.: Generalized data management system data structures and their mapping to physical storage. Ph.D. Th., University of Michigan, 1971.

[Taylor 1974]

Taylor, R. W.: "Data Administration and the DBTG Report", ACM SIGMOD Workshop on Data Description, Access, and Control, May 1974.

[Taylor and Frank 1976]

Taylor, R. W., and Frank, R. L.: "CODASYL Data-Base Management Systems", ACM Computing Surveys, Vol. 8, No. 1, March 1976.

[Tennent 1973]

Tennent, R. D.: "Mathematical Semantics and Design of Programming Languages", Tech. Report 59, Department of Computer Science, University of Toronto, September 1973.

[Tennent 1975]

Tennent, R. D.: "The Denotational Semantics of Programming Languages", Department of Computing and Information Science, Queen's University, Kingston, Ontario, April 1975.

[Tinman 1976]

Requirements for High Order Computer Processing Languages, "Tinman", Department of Defense, June 1976.

[TRW 1973a]

TRW: "SEMANOL Specification of JOVIAL", TRW Systems Group, RADC Contract No. F30602-72-C-0047, 26 March 1973.

[TRW 1973b]

TRW: "SEMANOL Reference Manual", TRW Systems Group, RADC Contract No. F30602-72-C-0047, 23 April 1973.

[Tsichritzis 1975]

Tsichritzis, D. A network framework for relational implementation Data Base Description, Proceedings of the IFIP TC-2 Special Working Conference on Data Base Description, Douque and Nijssen, eds., Elsevier-North Holland Publication Co., 1975, pp. 269-282.

[Tsichritzis and Lochovsky 1976]

Tsichritzis, D. C. and Lochovsky, F.: "Data Base Management Systems", Academic Press, 1976.

[Turn and Sine 1973]

Turn, Rein, and Sine, Barry: "Air Force Command and Control Information Processing in the 1980's: Trends in Hardware Technology", January 1973.

[Turn 1975]

Turn, Rein: "Computer Systems Technology Forecast", January 1975.

[USAS X3.9-1966]

USA Standard FORTRAN, United States of America Standards Institute, 1966.

[Uzgalis et. al. 1973]

Uzgalis, Robert, et. al.: "A UDL Definition of ALGOL 68", (Preliminary Draft), Computer Science Department, UCLA, 1973.

[van Wijngaarden 1966]

van Wijngaarden, A.: "Recursive Definition of Syntax and Semantics", in [Steel 1966], pp. 13-24.

[van Wijngaarden et. al. 1969]

van Wijngaarden, A. et. al.: "Report on the Algorithmic Language ALGOL 68", MR 101, Mathematisch Centrum, Amsterdam, 1969.

[van Wijngaarden et. al. 1973]

van Wijngaarden, A. et. al.: "Revised Report on the Algorithmic Language ALGOL 68", IFIP Press, 1973.

[van Wijngaarden et. al. 1975]

van Wijngaarden, A., Mailloux, B. J., Peck, J. E.L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L, G. L. T., and Fisker, R. G. (eds.): "Revised Report on the Algorithmic Language ALGOL 68", Acta Information, Vol. 5, 1-3, 1975.

[Varian 1969]

Varian 520/i Computer Handbook. Varian Data Machines, Irvine, California, November 1969.

[Waghorn 1975]

Waghorn, W.J. The DDL as an industry standard? Data Base Description, 121-168. Proceedings of the IFIP TC-2 Special Working Conference on Data Base Description, Douque and Nijssen, ed., Elsevier-North Holland,Publishing Co., 1975.

[Wegbreit 1971]

Wegbreit, Ben: "The ECL Programming System", Fall Joing Computer Conference, 1971, pp. 253-262.

[Wegner 1968]

Wegner, Peter: "Programming Languages, Information Structures, and Machine Organization", McGraw-Hill, New York, 1968, pp. 180-227.

[Wegner 1972]

Wegner, Peter: "The Vienna Definition Language", Computing Surveys, Vol. 4, No. 1, March 1972, pp. 5-63.

[Weinberg 1971]

Weinberg, G. M.: "The Psychology of Computer Programming", Van Nostrand, 1971.

[Welch 1976]

Welch, Terry A.: "An Investigation of Descriptor Oriented Architecture", Department of Electrical Engineering, University of Texas at Austin, Austin, Texas.

[Wichmann 1976]

Wichmann, B. A.: "Ackermann's Function: A Study in the Efficiency of Calling Procedures", BIT 16:1, 1976, pp. 103-110.

[Wilner 1971]

Wilner, Wayne: "Declarative Semantic Definition", STAN-CS-71, Computer Science Department, Stanford University, August 1971.

[Wilner 1972a]

Wilner, W., "Design of the Burroughs B1700", Fall Joint Computer Conference, 1972, pp. 489-497.

[Wilner 1972b]

Wilner, W.: "Burroughs B1700 Memory Utilization", Fall Joint Computer Conference, 1972, pp. 579-586.

[Wirth and Weber 1966]

Wirth, N., and Weber, H.: "EULER: A Generalization of ALGOL, and Its Formal Definition", Communications of the ACM, Vol. 9, Nos. 1, 2, 11 (Jan., Feb., Dec.) 1966, pp. 12-23, 89-99, 878.

[Wirth 1971]

Wirth, N.: "Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14, No. 4, April 1971, pp. 221-227.

[Wirth 1976]

Wirth, N.: "MODULA: A Language for Modular Multiprogramming, Institute fur Informatik, Report No. 18, Eidgenossische Technische Hochschule, Zurich, 1976.

[Wulf 1974a]

Wulf, W. A.: "ALPHARD: Towards a Language to Support Structured Programs", Computer Science Department, Carnegie-Mellon University, 1974.

[Wulf 1974b]

Wulf, W. A.: "C.mmp: A Multi-Mini-Processor", INFOTECH State of the Art Report 17: Computer Design, Infotech Information, 1974, pp. 585-600.

[Yamaguchi and Mertan 1974]

Yamaguchi, K. and Mertan A.G. Methodology for transferring programs and data. ACM SIGMOD Workshop on Data Description, Access and Control, May 1974, pp. 141-156.